Java™ magazine

By and for the Java community

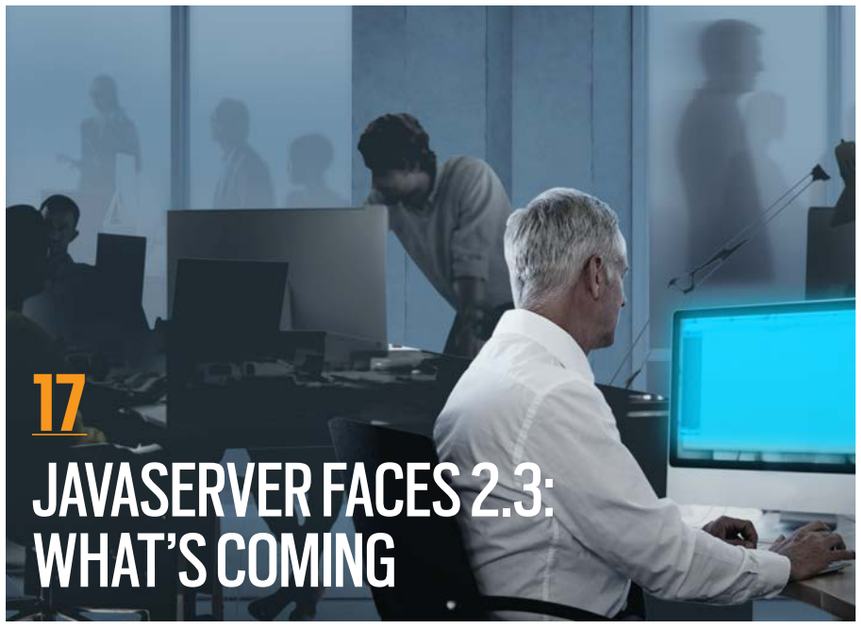
JULY/AUGUST 2016

ENTERPRISE JAVA

ORACLE.COM/JAVAMAGAZINE

ORACLE®

//table of contents /



COVER ART BY I-HUA CHEN

EDITORIAL

Editor in Chief
Andrew Binstock
Managing Editor
Claire Breen
Copy Editors
Karen Perkins, Jim Donahue
Technical Reviewer
Stephen Chin

DESIGN

Senior Creative Director
Francisco G Delgadillo
Design Director
Richard Merchán
Senior Designer
Arianna Pucherelli
Designer
Jaime Ferrand
Senior Production Manager
Sheila Brennan
Production Designer
Kathy Cygnarowicz

PUBLISHING

Publisher
Jennifer Hamilton +1.650.506.3794
Associate Publisher and Audience Development Director
Karin Kinnear +1.650.506.1985
Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES

Sales Director
Tom Cometa
Account Manager
Mark Makinney
Account Manager
Marcin Gamza
Advertising Sales Assistant
Cindy Elhaj +1.626.396.9400 x 201
Mailing-List Rentals
Contact your sales representative.

RESOURCES

Oracle Products
+1.800.367.8674 (US/Canada)
Oracle Services
+1.888.283.0591 (US)

ARTICLE SUBMISSION

If you are interested in submitting an article, please email the editors.

SUBSCRIPTION INFORMATION

Subscriptions are complimentary for qualified individuals who complete the subscription form.

MAGAZINE CUSTOMER SERVICE

java@halldata.com **Phone** +1.847.763.9635

PRIVACY

Oracle Publishing allows sharing of its mailing list with selected third parties. If you prefer that your mailing address or email address not be included in this program, contact Customer Service.

Copyright © 2016, Oracle and/or its affiliates. All Rights Reserved. No part of this publication may be reprinted or otherwise reproduced without permission from the editors. *JAVA MAGAZINE* IS PROVIDED ON AN "AS IS" BASIS. ORACLE EXPRESSLY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS OR IMPLIED. IN NO EVENT SHALL ORACLE BE LIABLE FOR ANY DAMAGES OF ANY KIND ARISING FROM YOUR USE OF OR RELIANCE ON ANY INFORMATION PROVIDED HEREIN. Opinions expressed by authors, editors, and interviewees—even if they are Oracle employees—do not necessarily reflect the views of Oracle. The information is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle. Oracle and Java are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Java Magazine is published bimonthly and made available at no cost to qualified subscribers by Oracle, 500 Oracle Parkway, MS OPL-3A, Redwood City, CA 94065-1600.

JavaOne™

September 18–22, 2016 | San Francisco
oracle.com/javaone

REGISTER NOW
Respond Early, Save $400*

JavaYour (Next)

- Learn about Java 9 and beyond
- 450+ educational sessions
- Explore innovations at the Java Hub
- Network with 500+ Java experts

Innovation Sponsor: intel
Diamond Sponsor: IBM
Gold Sponsor: Atlassian
Silver Sponsors: GitHub, New Relic.
Bronze Sponsors: Couchbase, Lightbend

*Save $400 over onsite registration. Visit oracle.com/javaone/register for early bird registration dates and details.

Copyright © 2016, Oracle and/or its affiliates. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

ORACLE

XRebel

ZEROTURNAROUND

THE LIGHTWEIGHT JAVA PROFILER

TRY IT FREE NOW!

Get a free t-shirt!

The Problem of Writing Small Classes

The highly recommended best practice presents a series of challenges that its advocates rarely address.

There are few coding practices that, when followed conscientiously, deliver as much benefit as writing small classes. Take almost any desirable metric—simplicity, testability, debuggability—and small classes score high. Take almost any undesirable metric—complexity, error density, brittleness—and small classes score low. In addition, various programming rules of thumb point directly toward smaller classes: the single-responsibility principle, in particular, as well as many fundamental refactorings.

With all these benefits, it seems that if you want to write good code, small classes should represent a fundamental implementation goal and class size should be a metric that is constantly kept in mind as code is written.

If you work this way, though, you'll find that for all the advice available on how to write good code and the wisdom of consultants, there is little guidance on how to manage the issues that small classes present.

Small classes have been a favorite concern of mine for a long time. I've written about how to tease small ones from larger ones, how to think in terms of small code units, and so on. But over the years, I have found that small classes—while delivering the promised benefits—create problems whose solutions are largely unexplored.

Let's begin by defining a small class. I define it as fewer than 60 lines of code (LOCs). The number appears arbitrary, but it works in delivering classes that can be seen in their entirety with a single page-up or page-down keystroke in the IDE. This same logic drives NASA's "Power of Ten" rule that limits functions also to 60 lines. Classes this size can be read and under-

PHOTOGRAPH BY BOB ADLER/GETTY IMAGES

stood quickly. Sixty LOCs is not a hard limit, but beyond it, I start getting itchy. Almost none of my classes exceed 70 LOCs. Some developers find a numerical limit objectionable and would prefer "the minimum LOCs necessary for the task"—which invariably leads to classes in which size is no longer a disciplined constraint—while a hard number leads you to review lengthy code looking for refactorings that simplify it.

Whatever number you settle on, if you start using it as a discipline-inducing limit, you quickly run into several challenges. The first is perhaps the most persistent problem in computing: naming. As objects become smaller, more are created, and more names must be formulated. FontHandler becomes FontLocator, FontVerifier, FontLoader, FontMetricExtractor, and so on. After a while, you begin to codify a set of naming conventions that you use with precision so that two classes with similar names can be readily distinguished. "Inspector" is not the same as "verifier," which is not the same as "validator," and so on.

This leads to a second problem, which is the proper grouping of small classes so that it's apparent they go together. This is where the lack of useful guidelines on designing Java packages becomes glaringly obvious. There are very few useful ideas on the topic of packaging. Sites use mostly a seat-of-the-pants approach, which is rarely the right way to do things. The trees in my package hierarchy are bushier than those of most projects. This provides benefits in that it's much clearer where to look for certain functionality. In addition, access restriction can be made very granular. The disadvantage is that it's not always easy to group classes into smaller packages. Choosing whether FontDisplay goes here or is more logically part of another package can take careful consideration. However, that kind of review helps refine the project's design.

The final problem is purely mechanical. In my IDE, I often have many, many tabs open to small classes, and I do a lot of bounding around between the various windows. At times, this can be a pain. If all the code were in one big class, I'd have one place to go, which is easy. However, when I got there, I would find myself constantly scrolling up and down, setting bookmarks, and jumping about inside that one big class. So, in effect, with small classes I've replaced that physical activity with my own. I use multiple windows across two screens, which solves many of my problems—at a glance, I can see the code I need in the open window.

In sum, working hard to create small classes presents a series of challenges that are little discussed either in the literature or in the counsel of experts. Given that diminutive classes make testability much easier and more thorough, facilitate legibility and maintainability, and enforce object-oriented basics, the extra work to figure out these challenges by yourself seems well worth it.

Andrew Binstock, Editor in Chief
javamag_us@oracle.com
@platypusguy

P.S. In this issue, we continue our refinement of the magazine's design with a more legible code font that lets us print more characters per line, and so wrap code less often. Feel free to send other suggestions to me at the address above.

MARCH/APRIL 2016

Polymorphic Dispatch with Enums

I would like to comment on the article "Making the Most of Enums" (March/April 2016, page 40). In that article, Michael Kölling points out how enums improve type safety and internationalization and how they allow the easy creation of thread-safe singletons.

While he mentions that "enum declarations are classes, and enum values refer to objects," he missed the opportunity to show how this enables polymorphism and allows for more object-oriented and maintenance-friendly programs.

Building upon the adventure game example, suppose you want to add the "drop" command. You have added the `DROP("drop")` constant to the enum and implemented the `dropItem()` method. Yet the program still does not recognize the command. The problem is that you failed to add the appropriate case to the `switch` statement.

Let us extend the enum declaration further:

```
public enum CommandWord {
    GO("go") {
        @Override
        public void exec(String secondWord) {
            // logic from the goRoom() method
        }
    },
    // ...
    // the other commands follow a similar pattern
    // ...
    QUIT("quit") {
        @Override
        public void exec(String secondWord) {
            // logic from the quit() method
        }
    };
    private String commandString;

    CommmandWord(String commandString) {
        this.commandString = commandString;
    }

    public String toString() {
        return commandString;
    }

    public abstract void exec(String secondWord);
}
```

By adding curly braces after the declaration of an enum constant, you create an anonymous subclass of `CommandWord`, which is used only to create the instance for this specific constant. By overriding the abstract `exec()` method, you add command-specific behavior to the individual constants.

With this change in place, the entire switch statement can be replaced with a single line:

```
commandWord.exec(secondWord);
```

If you now want to add the drop command, you only have to add another constant to the enum declaration. And if you forget to override the `exec()` method, the compiler will complain about it because it was defined as abstract in the `CommandWord` enum itself.

—Tobias Stensbeck

Michael Kölling responds: You make a very good point, and I did indeed miss an opportunity to go further and discuss polymorphic dispatch with enum methods, and how this can further improve the code. The gain you describe—avoiding the switch statement and replacing

it with a single polymorphic call—is one of the most significant improvements that come out of the replacement of constants with enums. It improves maintainability and removes the implicit coupling. You discuss the method and advantages perfectly, and all that remains for me is to thank you for bringing this up.

The Misery of Project Hosting?

In "The Miserable Business of Hosting Projects" (May/June 2016, page 4), Andrew Binstock speculates on whether the current model of free open source hosting is sustainable, examining several hosting services, including GitLab. While I can't speak for other companies, I can share thoughts from us at GitLab on several of the points mentioned in the piece.

In regard to the industry as a whole, it has seen some growing pains but it's not quite as "miserable" as the title of the article suggests. The freemium model is a well-known pricing strategy for SaaS [software-as-a-service]–based companies across many markets, not just hosting projects, with companies that have free solutions also offering paid versions to help customers pay only for what they need, along with allowing them to test-drive services.

We see this pricing strategy following the same path as email, which now exists as a free service with additional features (extra storage) available at an additional cost. In terms of cost efficiency, we've seen success through an open core model with a strong community vital to the development and implementation of new features. Other companies take a similar approach, taking revenue from an "Enterprise" or paid version of a service in conjunction with additional paid options to keep up with the growth in subscriptions for their free editions. Mr. Binstock mentioned that services currently free to developers using hosting services will eventually have to be paid for, but we see a future for free baseline features.

Although companies involved with hosting projects have either discontinued them or shut their doors, hosting projects will remain a vital, if not ubiquitous, part of the developer community as the demand for digital content and increased collaboration continues to grow.

—Job van der Voort
Vice President of Product, GitLab

Codehaus and What Came Next

Re: the editorial "The Miserable Business of Hosting Projects," I consider my involvement with the former Codehaus to be one of the defining aspects of my professional career. As folks who participated in the Codehaus know, the members were known as "hausmates" and many personal relationships were formed through the involvement of project participants. These relationships form a web of many of the bright stars of the industry, across ThoughtWorks, Walmart, DRW, Google, Square, Twitter, Red Hat, and others.

The Codehaus always tried to be pragmatic, in order to offer the best environment possible. This allowed us to take support from Atlassian for its suite of tools; from JetBrains for licenses to its IDEs; and others including Sonar, which grew at the Codehaus itself. Additionally, we were always thankful for the support of Matthew Porter and his company, Contegix. I'd be completely remiss if I did not mention Ben Walding, who kept the Codehaus operational for much of its lifetime. (I was merely a figurehead.)

The article rings true regarding how GitHub ate everyone's lunch. I love GitHub, and it has knocked the ball out of the park. While the footprint of its

services is smaller than that of the Codehaus, it's augmented by other organizations, such as Travis CI, CloudBees, and Google Groups. One of the leading points of the Codehaus Manifesto was acknowledging strong project leadership. The current combination of GitHub/GitLab, providers of continuous integration, and so on deliver such leadership within a project.

—Bob "The Despot" McWhirter
Cofounder, Codehaus

Erratum

In the May/June issue, in Mr. Kölling's article ["Understanding Generics," page 45], he several times uses `HashSet()` in his code examples. But I believe that he meant to use `HashMap()`, which will actually work in the code he presents.

—Bibhaw Kumar

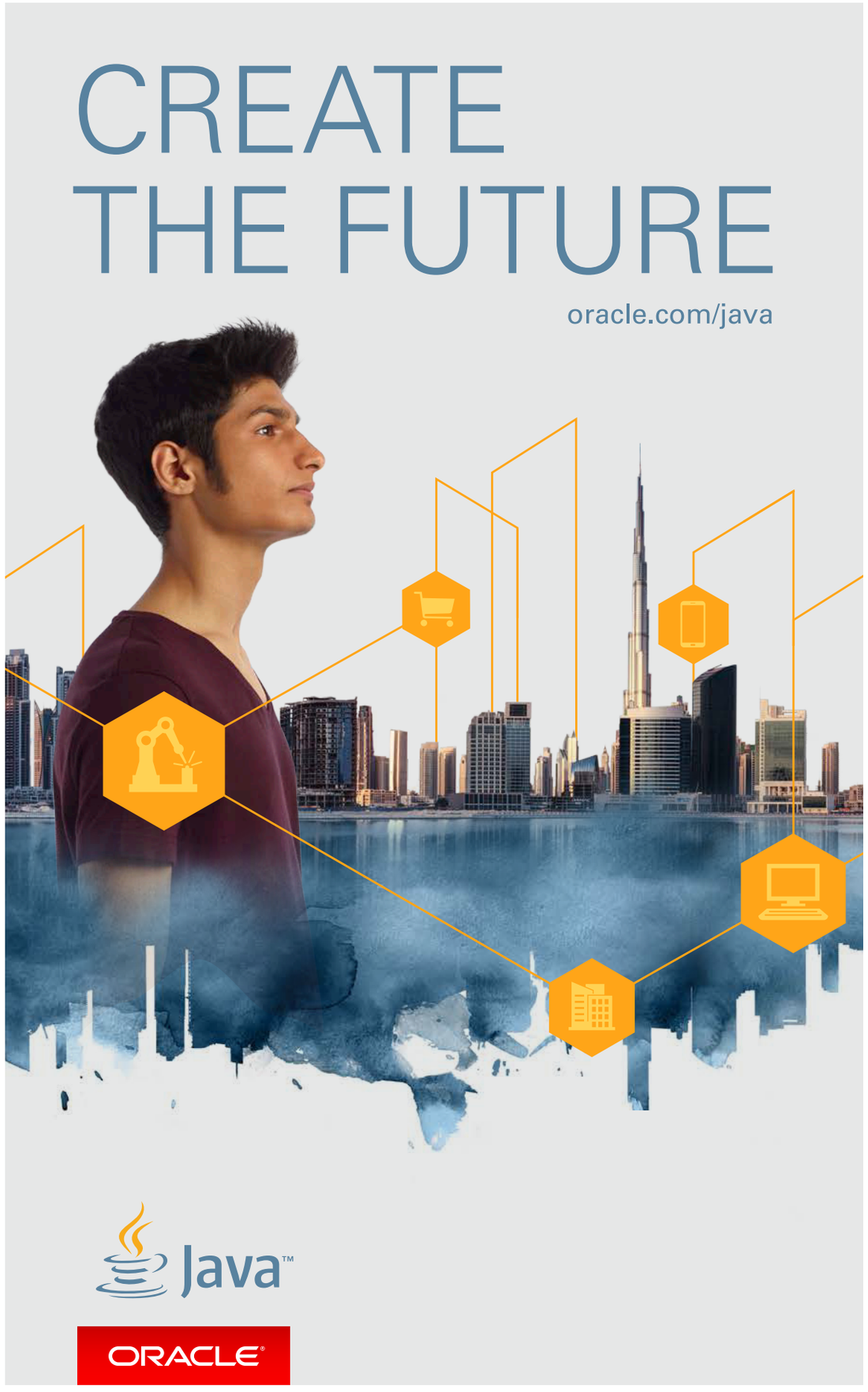
Michael Kölling responds: You're quite right. `HashSet()` *has only one generic parameter. My apologies for the confusion this caused.*

Where Are the Back Issues?

Several readers have inquired about the lack of access to back issues. This is a temporary problem that occurred when we switched content delivery networks. It should be resolved by press time or shortly thereafter. Our apologies for the inconvenience.

Contact Us

We welcome comments, suggestions, grumbles, kudos, article proposals, and chocolate chip cookies. All but the last two might be edited for publication. If your note is private, indicate this in your message. Write to us at javamag_us@oracle.com. For other ways to reach us, see the last page of this issue.



//events /

JavaOne *SEPTEMBER 18–22*
SAN FRANCISCO, CALIFORNIA
The ultimate Java gathering celebrates its 20th year. JavaOne features hundreds of sessions and hands-on labs. Topics covered include the core Java platform, security, DevOps, IoT, scalable services, and development tools. Georges Saab, vice president of development for the Java Platform Group at Oracle and chair of the OpenJDK governing board, and Mark Reinhold, chief architect of the Java Platform Group, are slated to speak at the event, as are many members of the Java development team. Highly anticipated Java 9 release enhancements will be presented and discussed. (See page 11 for more information.)

PHOTOGRAPH BY ERIC E CASTRO/FLICKR

JVM Language Summit
AUGUST 1–3
SANTA CLARA, CALIFORNIA
The JVM Language Summit is an open technical collaboration among language designers, compiler writers, tool builders, runtime engineers, and architects who target the JVM. This year's event will be held in Oracle's auditorium. Presentations will run in a single track and are allotted 45 minutes each (including questions). Workshop sessions will run for 60 minutes, with two or more sessions in parallel. Breakfast and lunch are served onsite. Breakout rooms are available for workshops, conversation, and ad hoc consultations. Presentations will be recorded and made available to the public.

DevOps Week DC
AUGUST 15–19
WASHINGTON DC
DevOps Week features a series of specialized courses designed to help organizations create an environment where the building, testing, and releasing of software can happen more rapidly and more reliably. Subject matter is aimed at software developers, engineers, project managers, quality assurance specialists, and test managers. Participants get both one-on-one interaction with instructors and opportunities to network with other software professionals.

JavaZone
SEPTEMBER 6–8
OSLO, NORWAY
This year marks the 15th anniversary of JavaZone. The event consists of a day of workshops followed by two days of presentations and more workshops. Last year's event drew more than 2,500 attendees and featured 150 talks covering a wide range of Java-related topics.

JDK IO
SEPTEMBER 13–15
COPENHAGEN, DENMARK
This annual event hosted by the Danish Java User Group consists of a two-day conference followed by one day of workshops. The focus is on all things Java: the language, the platform, the frameworks, and the virtual machine.

PHOTOGRAPH BY CCHANA/FLICKR

Strange Loop
SEPTEMBER 15–17
ST. LOUIS, MISSOURI
Strange Loop is a multidisciplinary conference that brings together developers and thinkers in fields such as emerging languages, alternative databases, concurrency, distributed systems, security, and the web.

JAX London
OCTOBER 10–12
LONDON, ENGLAND
JAX London is a three-day conference for cutting-edge software engineers and enterprise-level professionals, bringing together the world's leading innovators in the fields of Java, microservices, continuous delivery, and DevOps. Hands-on workshops take place on October 10, followed by conference sessions, keynotes, and expo.

O'Reilly Software Architecture Conference
OCTOBER 19–21
LONDON, ENGLAND
This year, the O'Reilly Software Architecture Conference is exploring evolutionary architecture to reflect the broadening of the field, encompassing new disciplines such as DevOps. Topics include strategies for meeting business goals, developing leadership skills, and making the conceptual jump from software developer to architect.

VOXXED Days THESSALONIKI
OCTOBER 21
THESSALONIKI, GREECE
The inaugural VOXXED Days event in Thessaloniki is a developer conference that promises expert speakers, core developers of popular open source technologies, and professionals willing to share their knowledge and experiences. Former Oracle Technology Evangelist Simon Ritter is scheduled to present "JDK 9: Big Changes to Make Java Smaller."

Devoxx
NOVEMBER 7–11
ANTWERP, BELGIUM
Devoxx is one of the largest mostly Java conferences in the world, with numerous experts from the US and Europe presenting a wide range of sessions on all aspects of Java development.

W-JAX
NOVEMBER 7–11
MUNICH, GERMANY
W-JAX is a conference focused on Java, architecture, and software innovation. More than 160 presentations on technologies and languages ranging from Java, Scala, and Android, to web programming, agile development models, and DevOps are planned. The main conference takes place November 8–10, with workshops scheduled on November 7 and 11. (No English page available.)

Special Note: Event Cancellation

QCon Rio
OCTOBER 5–7
RIO DE JANEIRO, BRAZIL
The organizers report: "Faced with an unstable political and economic environment . . . we considered it prudent to cancel our edition of QCon 2016. We emphasize that this decision does not affect the preparation of QCon São Paulo 2017."

Have an upcoming conference you'd like to add to our listing? Send us a link and a description of your event four months in advance at javamag_us@oracle.com.

JavaOne 2016

The largest Java conference is again a must-attend event.

JavaOne San Francisco, the central Java conference of the year, will be held September 18–22 at its customary site—a pair of hotels near Moscone Center in downtown San Francisco, California. As usual during the last 20 years of the conference, the first event is the opening keynote, on Sunday, September 18. The next four days see in-depth tutorials in the morning and more than 300 presentations running from late morning until early evening. Evenings are set aside for Birds of a Feather meetings, which are informal gatherings of developers who want to compare notes and share insights on a particular topic.

If you've ever attended JavaOne, you know that experts deliver the tutorials and presentations. Many of these experts are members of the core Java development team, Java Champions, or JavaOne Rock Stars. (This last title is accorded to JavaOne speakers whose previous sessions were among the most highly rated.)

The sessions in this year's event are divided into seven tracks on the following topics:

- **The core Java platform,** which has more than a dozen sessions dedicated to Java 9 and components of JDK 9 and half as many focused on illuminating the dark corners of Java 8
- **Emerging languages,** such as the JVM languages we cover in every issue—Kotlin, Groovy, Scala, and others—as well as cutting-edge languages that are emerging in new domains
- **Cloud and server-side development,** focusing on Java EE; enterprise technologies such as those discussed in this issue; and all things cloud, especially microservices
- **Devices,** including coverage of the Internet of Things and Java ME, among other topics
- **Java clients and user interfaces,** highlighted in 35 sessions, 20 of which are on JavaFX
- **Development tools**
- **DevOps and methodologies,** focusing mainly on automation but also featuring sessions on design, code quality, and project management

The conference is preceded on Saturday, September 17, by JavaOne4Kids, a series of workshops and tutorials for programmers and programmer-hopefuls from ages 10 to 18. On Sunday, September 18, there are full-day intensives, called Java University, which are separate events, paid for separately and held at a different venue. Finally, there is a trade-show component to JavaOne, with more than 40 vendors available to present products and answer questions.

JavaOne is a deep dive into all things Java and *the* place to meet and listen to the world's premier Java experts and speakers.

- Main JavaOne site
- Registration
- Catalog of sessions (viewable by track)

Oracle Press™

Your Destination for Java Expertise

Written by leading Java experts, Oracle Press books offer the most definitive, complete, and up-to-date coverage of Java available.

Raspberry Pi with Java: Programming the Internet of Things (IoT)

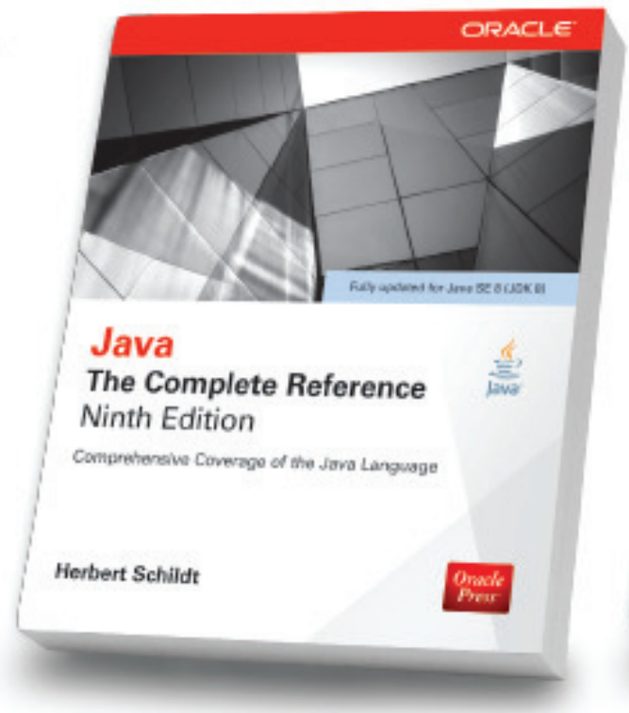
Stephen Chin, James Weaver

Use Raspberry Pi with Java to create innovative devices that power the internet of things.

Introducing JavaFX 8 Programming

Herbert Schildt

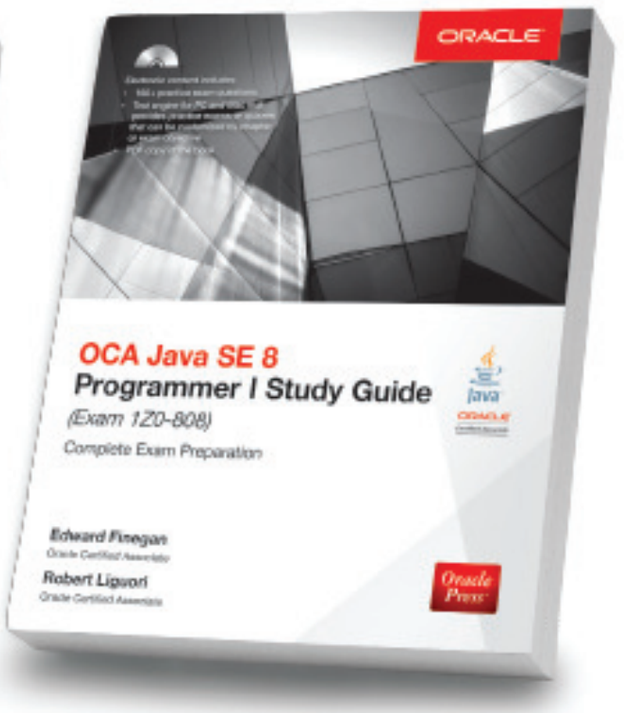
Learn how to develop dynamic JavaFX GUI applications quickly and easily.



Java: The Complete Reference, Ninth Edition

Herbert Schildt

Fully updated for Java SE 8, this definitive guide explains how to develop, compile, debug, and run Java programs.



OCA Java SE 8 Programmer I Study Guide (Exam 1Z0-808)

Edward Finegan, Robert Liguori

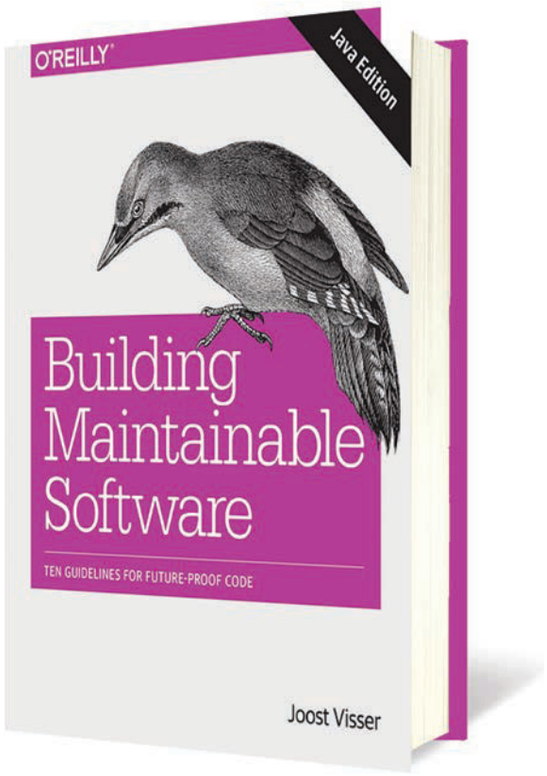
Get complete coverage of all objectives for Exam 1Z0-808. Electronic practice exam questions are included.

Available in print and as eBooks

Oracle Press™

www.OraclePressBooks.com • @OraclePress

//java books /



BUILDING MAINTAINABLE SOFTWARE (JAVA EDITION)

By Joost Visser (principal author)
O'Reilly Media

There are many books available on writing good code. They dig into a bag of well-known tips and recommendations that are aimed at beginner and intermediate programmers. The problem with some of these books is that it's hard to tell how the authors are qualified to dole out their advice. Most authors of these texts are consultants, which suggests that they see a wide range of code. However, most consultants work within a narrow range of industries and don't often stray into areas where programming is done substantially differently. For example, can authors credibly discuss rules for testability if they've never written software that could cost lives when errors occur? Can authors who have no experience proving code correct speak with authority on writing correct code?

One group of professionals comes close to meeting the high level of expertise to be qualified as authors: practitioners of software engineering, the discipline that quantifies software development via analysis of thousands of projects from all areas of programming. The problem is that most software engineering experts don't examine coding style and so, as a field, they've said relatively little on the topic. The principal author of this book and his colleagues come close to this level of qualification, though. For 15 years, they've run the Software Improvement Group, which studies software quality quantitatively based in part on coding style.

From this work, they came up with 10 guidelines, most of which will be familiar (write small units of code, use loose coupling, automate tests, and so on). They present them in the context of a prescriptive model of maintainability, which has strict numerical requirements for each guideline for code that aims to be in the topmost tier of quality. For example, on the rule against redundant code, the authors state that their model allows no more than 4.6 percent of lines of code to be redundant. That's helpful data, and it underscores the fact that guidelines cannot always be followed 100 percent of the time. I'll come back to this in a moment.

Each guideline is presented, its raison d'être explained, its application demonstrated, and the counterarguments to it contested. This last part is an imaginative addition that targets the reasons developers tell themselves for not obeying a guideline.

The examples chosen for each guideline contain problematic code and show the resolution, frequently relying on tried-and-true techniques. For example, to reduce duplicate code, the authors wisely suggest using the Extract Method and Extract

Superclass refactorings. These examples are useful although not sufficiently numerous to be a guide for readers who are keen to implement the suggestions fully. (For that, I recommend Martin Fowler's classic, *Refactoring*.)

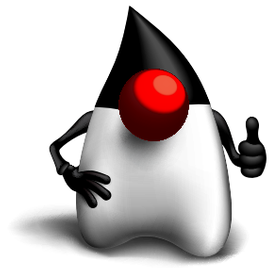
Some implementation suggestions strike me as dubious. For example, in the chapter on avoiding complexity—which is measured solely via cyclomatic complexity—the authors unwisely dig into the `switch` statement. The `switch` construct is a known weakness in the cyclomatic complexity measure, which greatly overestimates its complexity. In an example of associating colors with the flags of six European nations, the authors try to prove that a forgotten `break` statement in the six-way switch is the result of complexity. Problems that arise from complexity are generally the inability to understand code and how to fix it, rather than an omission that is detected by all code analyzers. The authors suggest that the ideal solution is to create six separate classes—one for each country—instantiate each one, and put the resulting objects into a `HashMap`.

This solution is suboptimal because the uniqueness of key values in a switch are no longer enforced in a `HashMap`. (Adding objects with the same key to a `HashMap` simply overwrites existing entries without any error.) The solution is also weak because if we handle the flags of, say, Africa, surely writing by hand 54 classes is more complex than one large `switch` statement. The proposed approach also shows a lack of understanding of `enums` in Java, which are full classes and guaranteed to maintain the unique keys. In addition, by using an `enum` declaration, all 54 classes for African flags are generated *by the compiler.* And the `EnumMap` gives you the data structure you want.

Although I wish the book more thoroughly explored topics and I have quibbles with some of the proposed solutions, the overall level of this work is better than many other volumes that advise how code should be written. It certainly can be recommended to beginners and early intermediate programmers and those whose work shows a repeated problem in code reviews. *—Andrew Binstock*

Get Java Certified

Oracle University



Upgrade & Save 35%*

- Get noticed by hiring managers
- Learn from Java experts
- Join online or in the classroom
- Connect with our Java certification community

Save 35% when you upgrade or advance your existing Java certification. Offer expires December 31, 2016. Click for further details, terms, and conditions.

ORACLE®

IJ

IntelliJ IDEA

CAPABLE AND ERGONOMIC JAVA* IDE

Get it now

or visit jb.gg/java

JET BRAINS

*Actually, much more than just Java

The Profusion of Enterprise Services

The history of Java in the enterprise is the story of the evolution of a complex knot of technologies into a palette of services that can be used collectively or individually. This evolution parallels the progress of services in general from tightly bundled to loosely coupled. This direction continues in the present with the design and implementation of so-called *microservices*.

In this issue, we examine some services that are used in enterprise apps either with containers or in full-scale Java EE apps. In the latter grouping is an update on JSF 2.3 (page 17), which is one of the most actively evolving standards. Our article on JavaMail (page 37) shows a classic Java EE service that can easily be used with other kinds of apps. Its value is not so much in building mail servers and readers but rather in enabling apps to send out alerts and updates to sysadmins or users.

JASPIC (page 25), the little-known but potent method of implementing custom security in applications, shows how much services can be created as standalone modules that plug into larger applications. Finally, for developers new to Java services, we include a tutorial on using JSON-P (page 31), the official libraries for handling JSON in Java.

We extend our series on JVM languages with an article on JRuby (page 62) written by its principal developer, Charlie Nutter. Our ongoing exploration of features in the upcoming Java 9 release examines JShell (page 43), the interactive REPL for Java. And, of course, we continue with our tutorials, detailed quiz, letters from readers, book review, and other content we expect you'll find interesting.

ART BY I-HUA CHEN

ARJAN TIJMS

JavaServer Faces 2.3: What's Coming

New features promise to resolve long-standing limitations.

JavaServer Faces (JSF) is the component-based model-view-controller (MVC) framework in Java EE. It was first included in Java EE 5 in 2006, although it had been available separately for two years prior to that. During its 12-year existence, JSF has reinvented itself several times.

JSF History up to Recent Times

In version 1.2, JSF transitioned from a separate framework to being integrated in Java EE, which led to fixing some major issues regarding JSP compatibility and removing JSF's own expression language (EL) in favor of the language provided by JSP.

JSF 2.0 in 2009 was the largest evolutionary step to date: postbacks, a heavy view state, and encapsulation of links (navigation rules) were all de-emphasized in favor of REST-style verbs, limited support for the MVC action pattern, and a much smaller view state (effectively abandoning the restore-view concept and instead rebuilding a view from scratch after a postback).

JSF 2.2, which appeared in 2013, continued the direction of JSF 2.0 by further de-emphasizing state with the introduction of a completely stateless mode, de-emphasizing components somewhat by introducing syntax to create pages directly in HTML (with only a special namespace attribute to connect the syntax to the server-side logic), and more support for the MVC action pattern.

While Contexts and Dependency Injection (CDI) had already transparently replaced the JSF native bean facility in JSF 2.0, some native features—such as the often-used view scope—kept people tied to native managed beans. JSF 2.2 started the alignment with CDI by introducing a CDI-compatible view scope as well as basing its new flow scope directly on CDI.

Were JSF created today, it would likely be based fully on CDI to begin with. That is, most of the factories and plugin points that JSF offers today would be based on the CDI bean manager, CDI extensions, and decorators. While this would certainly be desirable for new projects, such a full re-creation of JSF would be difficult, if not impossible, to keep backward-compatible. One of the virtues of JSF (and Java EE in general) is a strong focus on backward compatibility: 10-year-old JSF applications should still largely or even fully run on the very latest versions of Java EE. This often makes it relatively painless to upgrade. Instead of facing a large amount of up-front work in order to migrate to a newer version of Java EE, existing code can run "as is," while the application is updated piece by piece to take advantage of newer APIs.

In the light of this history, JSF 2.3 will align further with CDI, but it will do so in a backward-compatible way and provide switches for reverting back to earlier behavior. JSF 2.3 will also take advantage of Java 8 where possible and will take advantage of additional Java EE services, such as the WebSocket support that was introduced in Java EE 7.

In this article, I demonstrate two features that have largely been completed as of version 2.3 milestone 6 of the reference implementation. They are CDI alignment with regard to converters and validators, injection, and EL resolution; and using bean validation for multicomponent validation. I also show two features that are well underway but have not been completed at the time of writing this article. They are extensionless URLs, and programmatic and annotation-based configuration.

To follow this article, you need to be familiar with Java EE technologies and JSF concepts. It's not intended to be an introductory tutorial.

CDI Alignment

JSF has supported `@Inject`-based injection in many of its artifacts since JSF 2.2. This ability works much like how a servlet supports `@Inject` without actually being a CDI bean. There's a service provider interface (SPI) that each application server vendor needs to implement to provide injection services. In the case of Oracle's open source implementation of JSF, Mojarra, this is the `com.sun.faces.spi.InjectionProvider` interface.

While this approach provides an abstraction over the actual injection service, which is occasionally useful by itself, its disadvantage is obviously that it's a nonstandard interface that needs to be implemented separately for each JSF implementation and each Java EE application server that wishes to fully support JSF. This not only creates an N x M proliferation problem that prohibits freely mixing and matching implementations, but it also greatly limits the extent to which CDI features can be supported. For example, a `PhaseListener` may support `@Inject`, but it does not support a scope, can't be decorated, and can't contain interceptors, and the injection points generally can't be modified using CDI extensions.

For a limited number of artifacts (converters, validators, and behaviors), JSF 2.3 takes a different approach. A variant of those artifacts is available that does not rely on a proprietary SPI, such as the aforementioned `InjectionProvider`, but instead uses genuine CDI beans. Effectively, JSF uses the standardized `BeanManager` here as the mechanism to both obtain those artifacts and to provide injection and all the other services that come for free when CDI is used.

For backward-compatibility goals, the native lookup and injection machinery is retained. In fact, behind the scenes, Mojarra currently uses an old native converter that delegates to the CDI-based converter to transparently integrate this newer type of converter into the runtime. (This is a trick that's seen more often in Java EE.)

CDI-Based EL Resolver

Another area where JSF 2.3 replaces its own functionality with that of CDI is with respect to the EL resolver for implicit objects. Implicit objects are the variables that you can use via EL on, for example, a Facelet such as `#{facesContext}`, `#{request}`, `#{param}`, and so on. Currently the JSF spec states that these objects should be resolved via a JSF-specific EL resolver.

CDI, however, already provides a universal EL resolver. The main advantage of using this resolver is that JSF can automatically take advantage of specific performance improvements, and using the CDI resolver releases JSF from the burden of having to duplicate similar improvements and tunings in its own resolver.

The way the universal CDI EL resolver works is by having so-called built-in `Bean<T>`s available, which have a name (the implicit object's name) and a `create()` method that produces the implicit object itself. Almost a side effect from the point of view of EL resolving is the fact that by using `Bean<T>`, those very same implicit objects can also trivially be made available for injection. Furthermore, a scope can be set. Such scope will function as a kind of cache and might further help with performance, although care must be taken that the scope accurately reflects the lifetime of the implicit object.

The following code shows an example of a builder for a Bean<T>:

```
public class HeaderValuesMapProducer extends
         CdiProducer<Map<String, String[]>> {

   public HeaderValuesMapProducer() {
     super.name("headerValues")
       .scope(RequestScoped.class)
       .qualifiers(
          new HeaderValuesMapAnnotationLiteral())
       .types(
          new ParameterizedTypeImpl(
              Map.class,
              new Type[]{String.class,
                         String[].class}),
              Map.class,
                 Object.class)
       .beanClass(Map.class)
       .create(e ->
          FacesContext.getCurrentInstance()
                      .getExternalContext()
                      .getRequestHeaderValuesMap());
   }
}
```

Multicomponent Validation

One important reason for using a web framework such as JSF is that it provides well-defined facilities for validating data coming from the client. In JSF, this works by attaching either a native validator to the source side (the component, such as input text) or a bean validation constraint to the target side (the backing bean property).

In both cases, validation works only for input coming in via a single component. While this works great for validating that a password is at least eight characters, it doesn't help with the requirement that a password from the main input field be the same as one from the confirmation input field.

The need for multicomponent validation was recognized long ago and, in fact, the very first issue ever publicly filed for JSF asked for exactly this functionality. Historically, solutions to this problem were found in creating special components (such as a single component with two input fields, one for the main entry and one for the confirmation), using special multicomponent validators from utility libraries such as OmniFaces, or just validating manually in the action method.

In all this time, this basic problem was never addressed at a foundational level. JSF 2.3 has taken an initial attempt at resolving this problem by again utilizing an existing platform service: class-level bean validation.

The idea here is that a special constraint validator is attached to a backing bean. Per the bean validation rules, this attachment happens by first defining a special annotation, then defining an implementation of ConstraintValidator, then linking the annotation to the ConstraintValidator via an attribute on the annotation, and then annotating the backing bean with this.

The following code shows an example:

```
@Named @RequestScoped
@ValidIndexBean(groups =
    java.util.RandomAccess.class)
public class IndexBean implements
   ConstraintValidator<ValidIndexBean, IndexBean> {
   @Constraint(validatedBy = IndexBean.class)
   @Target(TYPE) @Retention(RUNTIME)
   public @interface ValidIndexBean {
       String message() default "Invalid Bean";
       Class<?>[] groups() default {};
       Class<? extends Payload>[] payload() default{};
   }
   public void initialize(
```

```
        ValidIndexBean constraintAnnotation) {}
    public boolean isValid(
        IndexBean other,
        ConstraintValidatorContext context) {
           return other.getFoo().equals(other.getBar());
    }

    @NotNull
    private String foo;

    @NotNull
    private String bar;
    // + getters/setters
}
```

For a reusable validator, such as `@Email`, which can be applied to many different fields in different beans, this work is surely worth it. Multicomponent validation for backing beans is, however, often more ad hoc, and for a one-off validation case for a single bean, the number of moving parts is perhaps a bit too much.

Alternatively, a library of somewhat more-reusable validators can be created—for example, the bean validation counterparts of the OmniFaces multicomponent validators such as `validateEqual`, `validateOneOrMore`, `validateOneOrNone`, and so on.

To contrast with the example above, I'll show an example of a reusable `validateEqual` validator that uses an EL-enabled attribute to specify the bean properties that should be validated. Note that using EL for this is just an example and there are certainly other feasible options, such as marking the properties with annotations.

I'll first define the validation annotation, this time separately:

```
@Constraint(validatedBy = ValidateEqualValidator.class)
@Target(TYPE) @Retention(RUNTIME)
public @interface ValidateEqual {
   String message() default "Invalid Bean";
   Class<?>[] groups() default {};
   Class<? extends Payload>[] payload() default {};
   String[] inputs();
}
```

Note the extra attribute `inputs`.

Then the actual validator can be defined:

```
public class ValidateEqualValidator implements
   ConstraintValidator<ValidateEqual, Object> {

   private List<String> inputs;

   public void
     initialize(ValidateEqual constraintAnnotation) {
       this.inputs =
         asList(constraintAnnotation.inputs());
   }

   public boolean
     isValid(Object bean,
             ConstraintValidatorContext ctx) {
       return new
         HashSet<>(collectValues(bean, inputs))
            .size() == 1;
   }
}
```

This validator works by leveraging the platform-provided `ELProcessor` from the EL 3.0 spec to easily obtain the property values from the bean that is being validated. This is done as follows:

```
public static List<Object> collectValues(Object bean) {
   ELProcessor elProcessor = getElProcessor(bean);

   return inputs.stream()
         .map(input -> elProcessor.eval(input))
         .collect(toList());
}
```

A fully functional `ELProcessor` for usage in a Java EE environment can be obtained by just instantiating a new instance, and then providing it with the `ELResolver` from the CDI bean manager. This can be done as shown below:

```
public static ELProcessor getElProcessor(Object bean) {
   ELProcessor elProcessor = new ELProcessor();

   elProcessor.getELManager()
         .addELResolver(
               CDI.current()
                  .getBeanManager()
                  .getELResolver());

   elProcessor.defineBean("this", bean);

   return elProcessor;
}
```

Notice, in particular, that in the `getElProcessor()` method, the bean to be validated is being added to the `ELProcessor` context under the `"this"` name. This approach will be used when the properties that should be validated are defined shortly. (As a side note, it's perhaps interesting to realize that four different Java EE specs are used together here rather seamlessly: those for JSF, bean validation, EL, and CDI.)

Finally, the backing bean is annotated again, but this time with the reusable validator:

```
@Named
@RequestScoped
@ValidateEqual(
  groups = RandomAccess.class,
  inputs={"this.foo", "this.bar"})
public class IndexBean {
   @NotNull
   private String foo;
   @NotNull
   private String bar;
  // + getter/setters
}
```

Note here that the `inputs` attribute is initialized with EL references to the two properties that should be validated together.

In both cases, the part of a Facelet containing the actual input components looks as follows:

```
<h:form>
  <h:inputText value="#{indexBean.foo}">
     <f:validateBean validationGroups =
       "javax.validation.groups.Default,
        java.util.RandomAccess" />
  </h:inputText>

  <h:inputText value="#{indexBean.bar}">
      <f:validateBean validationGroups =
        "javax.validation.groups.Default,
         java.util.RandomAccess" />
  </h:inputText>

  <f:validateWholeBean value="#{indexBean}"
      validationGroups="java.util.RandomAccess"/>

  <h:commandButton value="submit" />
</h:form>
```

[Lines 4 and 5 are wrapped due to space constraints and should be entered as a single line, as are lines 10 and 11. —*Ed.*]
A rather important aspect of full class bean validation in combination with JSF is that the bean seen by the validator is a *copy* of the backing bean and not the actual backing bean. The reason for this is that the JSF validation semantics demand that the model (the backing bean) is not updated when any validation or conversion failure happens. However, full class bean validation can happen only after the bean has been fully updated. To break this mismatch, the runtime first makes a copy of the backing bean, updates this copy, and validates it. Only if all validation constraints pass is the actual backing bean updated.

Extensionless URLs

JSF is implemented internally via a servlet, the so-called `FacesServlet`, which listens to requests that have a specific pattern. By default, this pattern includes `"/faces/*"`, `"*.jsf"`, `"*.faces"`, and, as of JSF 2.3, `"*.xhtml"`. Users can set their own pattern in web.xml using the same servlet syntax that's used to map any servlet to a URL pattern.

As can be seen from the information above, both path mapping and extension mapping are supported. An example of path mapping would be `http://example.com/faces/page.xhtml`, while an example of extension mapping would be something like `http://example.com/page.jsf`.

In modern web applications, it's often desirable to have "clean" URLs—that is, URLs that specifically don't have an extension and generally don't have any kind of clutter in them. Unfortunately, JSF does not support such URLs out of the box. Curiously, even when you are using path mapping, an extension is still required, as shown above.

Clean URLs in JSF can be obtained by using third-party libraries, such as PrettyFaces or OmniFaces, but this kind of functionality is now deemed to be sufficiently well understood and mature that it qualifies for inclusion in JSF itself.

The goal for extensionless URLs in JSF 2.3 is threefold:

First, path mapping should work without an extension. For example, a URL such as `http://example.com/faces/page` should work out of the box.

Second, exact mapping should be officially supported. For example, a URL such as `http://example.com/page` should work when the following mapping is present in web.xml:

```
<servlet>
   <servlet-name>Faces Servlet</servlet-name>
   <servlet-class>
      javax.faces.webapp.FacesServlet
   </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/page</url-pattern>
</servlet-mapping>
```

Third, the most ambitious part of the goal is that a URL such as `http://example.com/page` should work without requiring the user to map `/page` (or any page) explicitly by using exact mapping, but rather it should work by setting only a single configuration option.

At press time, it's not yet clear how the third goal should be implemented, but a possible approach would be adding a new `listViewResources()` method to the `ResourceHandler`, and then taking advantage of the Servlet 3.0 spec's programmatic mapping during application startup to automatically add exact mappings for all view resources (for example, Facelets) that are handled by a given `ResourceHandler`.

Programmatic and Annotation-Based Configuration

High-level declarative configuration in JSF is done using either context parameters in web.xml or the dedicated `faces-config.xml` files. Additionally, there are some

lower-level options available, such as configuration via the `Application` class or programmatically by providing the content of a `faces-config.xml` file via a callback and the XML Document Object Model (DOM) API.

Context parameters in web.xml particularly have the disadvantage of consisting of mere strings (usually long ones) that have to be looked up and can be misspelled easily. It's also not directly possible to see what the defaults are for any given configuration item.

Both web.xml and the `faces-config.xml` file that resides in a .war file's WEB-INF folder have the shared disadvantage that they cannot be read by CDI extensions and other artifacts that start up early in the Java EE boot process.

Finally, XML-based files (so-called deployment descriptors) are not particularly flexible, specifically because there's no overall platform service in Java EE that allows placeholders in them or a way to provide conditional included files or overlays.

Because of the above issues, it's planned to provide an annotation-based and optionally programmatic high-level configuration system in JSF. In its most basic form, such configuration looks as follows:

```
@FacesConfig
public class SomeClass {

}
```

This configuration by itself will automatically add the JSF servlet mappings, which is currently done when an empty (but valid) `faces-config.xml` file exists or when, for example, the deprecated `@ManagedBean` annotation is encountered on any class in the application.

As an alternative to the mentioned context parameters, attributes on the `@FacesConfig` annotation can be used to configure various aspects of JSF:

```
@FacesConfig(
   stateSavingMethod = Server,
   faceletsRefreshPeriod = -1,
   projectStage = Production
)
public class SomeClass {

}
```

The intent here is to use strongly typed values where possible. For example, the `Server` value would come from an enumeration.

Although it hasn't been fully worked out yet, the programmatic aspect *might* be added using EL-enabled attributes, much as in the validator example shown earlier, for example:

```
@FacesConfig(
   stateSavingMethod = Server,
   faceletsRefreshPeriodExpr =
       "this.faceletsRefreshPeriod",
   projectStageExpr = "configBean.dev?
       'Development' : 'Production'"
)
public class SomeClass {
   int getFaceletsRefreshPeriod() {
       return … ? -1 : 0;
   }
}
```

In this example, the `faceletsRefreshPeriod` is set by an expression that directly refers to a property of the bean on which the annotation appears. Inside the getter method of that property, arbitrary logic can be used to determine the desired outcome. The `projectStage`, however, is set

by an expression that performs some logic directly in EL itself. Although the complexity of such EL expressions can be fairly high, it's good practice to keep them fairly small. Note that referencing any bean other than "this" might not be supported for those attributes that have to be read by CDI extensions.

Conclusion

JSF 2.3 is moving forward by using existing services from the Java EE platform and providing glue code where necessary to bridge gaps. In addition to that main theme, an assortment of features will be introduced that aim to keep JSF up to date and generally easier to use. The features presented in this article represent a work in progress and might still change before the final release of JSF 2.3. More details about the features discussed here, as well as about other JSF 2.3 features, can be found on my blog. </article>

Arjan Tijms is a member of the Expert Groups for JSF (JSR 372) and the Java EE Security API (JSR 375). He is the cocreator of the popular OmniFaces library for JSF that was a 2015 Duke's Choice Award winner, and he is the main creator of a suite of tests for the Java EE authentication service provider interface (JASPIC) that has been used by several Java EE vendors. [See the article on JASPIC in this issue. —*Ed.*] Tijms holds a Master of Science degree in computer science from the University of Leiden in the Netherlands.

learn more

- The author's blog on JSF 2.3 updates
- The official JSF 2.3 JSR

13 Billion
Devices Run Java

ATMs, Smartcards, POS Terminals, Blu-ray Players, Set Top Boxes, Multifunction Printers, PCs, Servers, Routers, Switches, Parking Meters, Smart Meters, Lottery Systems, Airplane Systems, IoT Gateways, Programmable Logic Controllers, Optical Sensors, Wireless M2M Modules, Access Control Systems, Medical Devices, Building Controls, Automobiles...

Java™ | #1 Development Platform

ORACLE®

STEVE MILLIDGE

Custom Servlet Authentication Using JASPIC

A little-known Java EE standard makes it simple to enforce authentication using your preferred resources.

When you build web applications using Java EE, you often need to work with some organization-specific user repository for authenticating users and obtaining a user's groups. Typically users are defined in a specific database, a strange LDAP configuration, or some other user-identity store specific to the project. All Java EE application servers ship with the capability to integrate with a common set of identity stores. For example, GlassFish Server ships with several so-called *realms*: file, LDAP, JDBC, Oracle Solaris, PAM, and certificate.

Each realm needs to be manually configured, and the configuration is specific to the application server and outside the control of your application. If the predefined realms don't fit your needs, you then need to develop an application-specific module to extend the capabilities using application server–specific APIs. Many developers faced with this prospect build some custom code in the web application, which integrates with their required identity store and uses application-specific mechanisms to manage authentication and authorization.

The problem with this approach is that these developer-designed mechanisms for managing authentication are not integrated with the application server, so the standard Java EE security model does not apply, the power of Java EE APIs such as `isUserInRole` and `getUserPrincipal` can't be used, and standard Java EE declarative security fails. In this article, I examine an alternative solution that is tucked away in Java EE. I expect readers to have a basic working knowledge of Java EE and its authentication mechanisms.

Enter JASPIC

When developers design their own authentication modules, the Java Authentication Service Provider Interface for Containers (JASPIC) provides an elegant solution. JASPIC has been part of Java EE since Java EE 6, but it is not well known and has a reputation for being difficult to use. The goal of the JASPIC specification is to define, in a standard way, how the authentication process occurs within a Java EE container and the points within that process where custom authentication modules for validating security messages, users, and groups can be integrated.

If you just download the JASPIC specification and dive right in with the aim of building a compliant Server Authentication Module (SAM), you will surely become confused and dispirited. This is because the specification is designed to describe in depth what an implementer of a Java EE container has to do. It also covers both client and server authentication and a large number of security scenarios, most of which are not relevant to you.

In this article, I cut through the confusion and demonstrate that developing an authentication module that is well inte-

grated with Java EE and comes packaged in your web application is actually, barring some boilerplate code, pretty simple. This is because in the case of custom servlet authentication, JASPIC provides a small profile and defines the interaction between your module and the servlet container—and this interaction is fairly simple. In this article, I assume that you are familiar with standard Java, Java Authentication and Authorization Service (JAAS), and Java EE security concepts such as principals, subjects, and callback handlers.

SAM Concept

JASPIC defines in its message processing model (MPM) how authentication occurs in a container. The MPM defines the specific processing steps an inbound HTTP request into the servlet container progresses through to be validated and secured. At the heart of JASPIC is the concept of a SAM. A SAM is called at specific points in the processing of the servlet request, as shown in **Figure 1**.

As can be seen in **Figure 1**, the SAM's `validateRequest` method is called by the servlet container whenever there is an inbound servlet request, prior to the request being passed to the servlet for processing.

The SAM is also called after the servlet request is complete to enable additional postprocessing of the servlet response before it is returned to the client. The SAM is the key component you need to implement to develop a custom authentication provider for your web application. A SAM needs to implement the JASPIC-defined interface `ServerAuthModule`. **Listing 1** shows the key method definitions that need to be implemented.

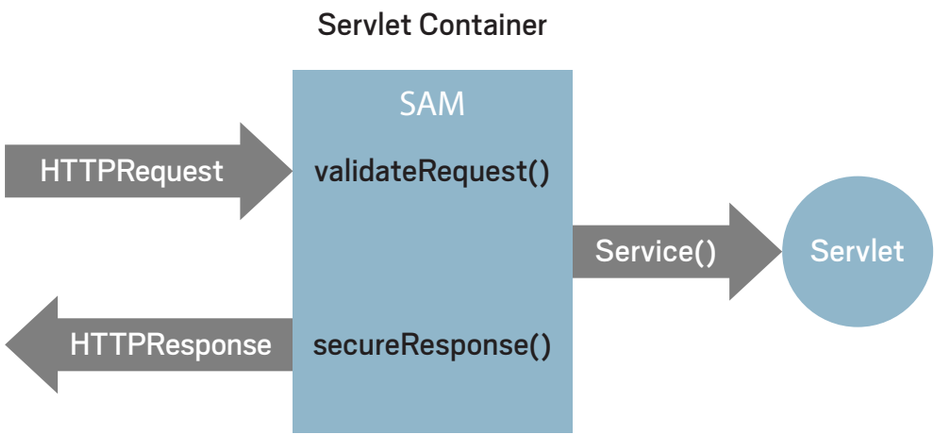


Figure 1. JASPIC servlet MPM

Listing 1.

```
public Class[] getSupportedMessageTypes();

public void initialize(
            MessagePolicy requestPolicy,
            MessagePolicy responsePolicy,
            CallbackHandler handler, Map options)
                throws AuthException;

public AuthStatus validateRequest(
            MessageInfo messageInfo,
            Subject clientSubject,
            Subject serviceSubject)
                throws AuthException;

public AuthStatus secureResponse(
            MessageInfo messageInfo,
            Subject serviceSubject)
                throws AuthException;

 public void cleanSubject(
            MessageInfo messageInfo,
            Subject subject)
                 throws AuthException;
```

I'll examine each method in turn. The `getSupportedMessageTypes` method indicates to the container the types of messages your module supports. For the example, in the case where you're authenticating servlet calls, you must return

HttpServletRequest and HttpServletResponse classes in your implementation, as shown in **Listing 2**.

Listing 2.

```
public Class[] getSupportedMessageTypes() {
    return new Class[] {
        HttpServletRequest.class,
        HttpServletResponse.class};
}
```

The initialize method of your SAM should configure the SAM based on the properties passed in. The options map could contain ServletContext initializer parameters, and these could be used to initialize your SAM with the properties required to access a database, an LDAP server, or any custom properties you need to set up your SAM. However, the key thing to do in this method is to store a reference to the passed-in CallbackHandler, because you will need it in your validateRequest implementation to pass the user and group principals to the servlet container. A simple implementation of initialize looks like **Listing 3**.

Listing 3.

```
public void initialize(
            MessagePolicy requestPolicy,
            MessagePolicy responsePolicy,
            CallbackHandler handler,
            Map options)
               throws AuthException {
                    this.handler = handler;
}
```

The secureResponse method is called after the servlet request has been processed. In the case of a simple SAM for use with servlets, this method doesn't really need to contain any specific processing. If you need to do any postprocessing of the servlet response at this point you can, but a simple implementation would be just to return success, as shown in **Listing 4**.

Listing 4.

```
public AuthStatus secureResponse(
           MessageInfo messageInfo,
           Subject serviceSubject)
                throws AuthException {
                    return SEND_SUCCESS;
}
```

Another method that is not really needed for developing a servlet authenticator but is part of the interface of the SAM is cleanSubject. This method can be implemented as a no-op, as shown in **Listing 5**.

Listing 5.

```
public void cleanSubject(
           MessageInfo messageInfo,
           Subject subject)
               throws AuthException {
}
```

Implementing validateRequest()

As you can see from the previous listings, the majority of the methods you need to implement for your SAM can be fairly straightforward. The final method to be implemented, validateRequest, is the heart of your authentication provider. The implementation of this method needs to perform several key tasks.

- Retrieve the servlet request and servlet response from the MessageInfo object and retrieve whatever information you need to use to authenticate a user.
- Connect to your identity store and authenticate the user and retrieve the groups associated with the user.

- Use the stored callback handler to pass these user and group principals to the servlet container.
- Finally, return success to the servlet container to allow the request to proceed to the servlet.

In **Listing 6**, which shows the body of my `validateRequest` method, I retrieve the username and the user's groups directly from the servlet request parameters and use these to set up the user and group principals. Obviously this is not very secure, but it illustrates the skeleton of what needs to be done.

Listing 6.

```
HttpServletRequest request =
    (HttpServletRequest)
        messageInfo.getRequestMessage();
String user = request.getParameter("user");
    String groups[] =
        request.getParameterValues("group");

Callback callbackArray [] = null;
if (user != null && groups != null ) {
    // callback used to set the user Principal
    Callback userCallback =
        new CallerPrincipalCallback(
            clientSubject, user);
    Callback groupsCallback =
        new GroupPrincipalCallback(
            clientSubject,groups);
    callbackArray = new Callback[] {
        userCallback,
        groupsCallback};
}
else {
    callbackArray = new Callback[] {
        new CallerPrincipalCallback(
            clientSubject,
                (Principal)null)
    };
}

try {
    handler.handle(callbackArray);
} catch (Exception ex) {
    AuthException ae =
        new AuthException(ex.getMessage());
    ae.initCause(ex);
}
return SUCCESS;
```

Some key points to note about this implementation are that if you decide that the authentication is successful, the resulting `Principal`s need to be passed to the container. To pass `Principal`s to the servlet container, you need to create instances of specific callbacks that are defined by JASPIC. The first is a `CallerPrincipalCallback`, which should be initialized with the `clientSubject` passed into your `validateRequestMethod` and a `String` representing your username or a custom `Principal` object.

The second callback is a `GroupPrincipalCallback`, which also should be initialized with the `clientSubject` and with an array of `String`s representing the names of the groups the user belongs to or an array of custom `Principal`s. These callback handlers are then passed to the `handle` method of the handler you stored in your `initialize` method earlier so that the servlet container can initialize the Java EE caller principal and set up the Java EE roles.

If the authentication is not successful, you need to create a `CallerPrincipalCallback` initialized with the `clientSubject` and a null `Principal`, and then pass these to the handler. This has the effect of letting the request proceed but with no user associated. Authorization security checks in the container will then deny access.

Therefore, you always return `SUCCESS` from your `validateRequest` method. `FAILURE` should be used as a return value only if there was some problem with your SAM—for example, if you were unable to contact some external resource such as an LDAP server.

Registering Your SAM

To deploy your custom SAM with your application, you package the implementation classes into the WAR file, as you would for any other application classes. The JASPIC specification defines how to register and unregister a custom SAM; this can be done in a `WebListener`, which is called when your web application starts. **Listing 7** shows how to register and unregister the SAM in a `WebListener`.

Listing 7.

```
@WebListener
public class SimpleSAMWebListener implements
    ServletContextListener {
    private String registrationid;

    public void contextInitialized(
        ServletContextEvent sce) {
            String appContext =
               registrationid =
               AuthConfigFactory.getFactory()
               .registerConfigProvider(
                  new SimpleSAMAuthConfigProvider(
                                       null,null),
                  "HttpServlet",
                  appContext,
                  "Simple SAM");
        }

    public void contextDestroyed(
        ServletContextEvent sce) {
            AuthConfigFactory
                .getFactory()
                .removeRegistration(
                    registrationid);
        }
}
```

There are some additional boilerplate classes required to integrate your SAM into the JASPIC infrastructure. These classes are implementations of three interfaces, and they are rarely different than the versions in the zip file available in the *Java Magazine* download area. Typically, if you're using a single SAM, you'll use the following files without modification:

- `AuthConfigProvider` is a factory for creating `ServerAuthConfig` objects.
- `ServerAuthConfig` is an object that describes a configuration for a specific application context and message layer—which, in the case of my servlet application, is always the same. `ServerAuthConfig` is also a factory for `ServerAuthContext` objects.
- `ServerAuthContext` is a class that wraps the SAM, because in the general case there can be multiple SAMs, but in most cases—and in my example—there is only one. If there are multiple SAMs, the `ServerAuthContext` implementation should call each in turn and then adjudicate the results.

The implementation included in this article will work as expected unless there are more-complex initialization and configuration requirements or there are multiple SAMs that need to be invoked.

Testing the Example SAM

In my example SAM, I implemented `validateRequest` so that the user and groups were obtained from the servlet request parameters.

To test the SAM, I need to define a servlet with a security constraint, as shown in **Listing 8**.

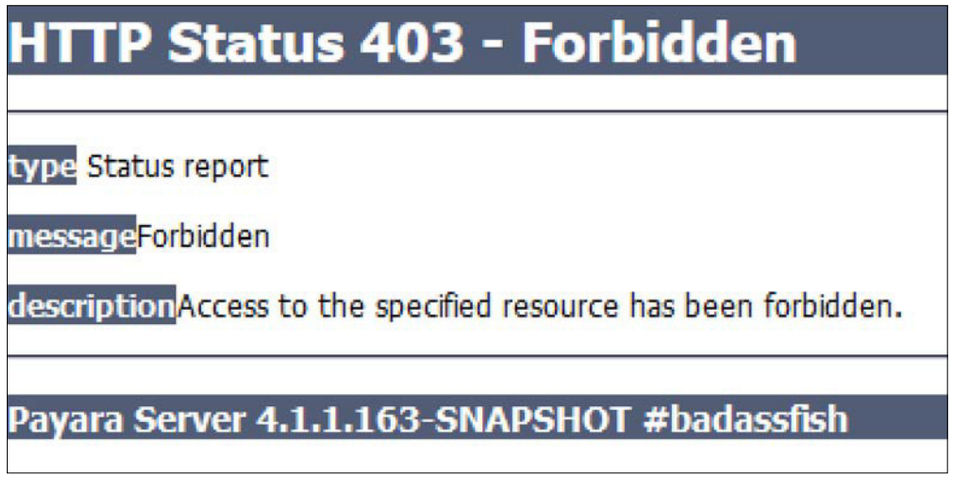


Figure 2. Testing error message

Listing 8.

```
@WebServlet(name = "SecureServlet",
            urlPatterns = {"/SecureServlet"})
@DeclareRoles("admin")
@ServletSecurity(@HttpConstraint(
            rolesAllowed = "admin"))
public class SecureServlet extends HttpServlet {
 ...
```

For this example, I implemented the servlet so that it just prints the caller `principal`:

```
 out.println("User Principal is " +
             request.getUserPrincipal().getName());
```

If I access the servlet directly without any request parameters, I receive the "forbidden access" response from the container, because my SAM cannot find the user or groups (see **Figure 2**).

If I use the URL—including a user and group admin, as in `http://127.0.0.1:8080//jaspic-sam-example/SecureServlet?user=steve&group=admin`—I get the authenticated response from the servlet (see **Figure 3**) because the SAM sets

Figure 3. Successful test result showing user data

up the admin role and the caller principal from the servlet request parameters.

For this to work in your application server, you will need to configure role mapping in your web application for the logical role admin that is declared on your servlet to be mapped to a server group admin. This is typically done in the application server–specific deployment descriptor. In the case of GlassFish, the server can be configured to map roles automatically to the group with the same name.

Conclusion

I encourage you to use JASPIC to build your own custom web application authentication modules. It is not too difficult once you get started and you realize that the core of the implementation is purely in your `validateRequest` method of your custom SAM. The additional support classes can be used directly from my example project to support your SAM and are sufficient for the majority of cases. Once you have built a SAM, you can take full advantage of the power of the standard Java EE declarative security mechanisms for securing your application. `</article>`

Steve Millidge (@l33tj4v4) is the founder and director of Payara Services and C2B2 Consulting. He has used Java extensively since version 1.0 and is a member of the Expert Groups for JSR 107 (Java caching), JSR 286 (portlets), and JSR 347 (data grids). He founded the London Java EE User Group and is an ardent Java EE advocate. Millidge has spoken at several conferences.

DAVID DELABASSÉE

Using the Java APIs for JSON Processing

Two easy-to-use APIs greatly simplify handling JSON data.

JavaScript Object Notation (JSON) enables lightweight data interchange. It is often used in lieu of XML, but clearly both options have their benefits and drawbacks. XML is powerful, but its power comes at the price of complexity. On the other hand, JSON is somewhat more limited than XML but this leads to one of the main benefits of JSON: its simplicity. This simplicity probably explains why today, JSON is unarguably the most common data interchange format on the internet. JSON is often associated with REST services, but traditional enterprise applications are more and more using JSON, too, so the introduction of JSON in the latest version of Java EE—Java EE 7—was a welcome addition to the platform.

JSON support is delivered through the new Java API for JSON Processing (JSON-P), which was standardized in JSR 353. This specification defines a simple API to process—that is, parse, generate, transform, and query—JSON documents. Note that binding (that is, marshaling of Java objects to JSON documents and vice versa) will be addressed in a related API, the Java API for JSON Binding (JSON-B), which is currently being defined in JSR 367.

JSON-P offers not one but two APIs: a high-level object model API that is similar to the XML Document Object Model (DOM) API and a lower-level streaming API that is similar to the Streaming API for XML (StAX). This article provides a brief introduction to both these APIs.

The JSON-P Object Model API

The JSON-P object model API is based on an in-memory, tree-like structure that represents the JSON data structure in a way that can be queried easily. The API also enables navigation through this JSON tree structure. Note that this API delivers ease of use, but it consumes more memory and is not as efficient as the lower-level streaming API, which I discuss later in this article.

The object model API supports the different JSON data types via the following classes: `JsonObject`, `JsonArray`, `JsonString`, and `JsonNumber`. In addition, the class `JsonValue` defines a few constants to represent specific JSON values (true, false, and null). `JsonValue` is also the common supertype of `JsonObject`, `JsonArray`, `JsonString`, and `JsonNumber`.

The object model API resides in the `javax.json` package and works with two principal interfaces: `JsonObject` and `JsonArray`. `JsonObject` provides a `Map` view for accessing the unordered collection of zero or more key-value pairs representing the model. `JsonArray` provides a `List` view for accessing the ordered sequence of zero or more values of the model. Using the `JsonReader.readObject` method, you can create instances of either type from an input source. You can also build `JsonObject` and `JsonArray` instances using a fluent API, as I explain next.

To create a model that represents a JSON object or a JSON array, the object model API relies on a simple builder pattern.

You just need to use static methods from the `Json` class (`Json.createObjectBuilder` or `Json.createArrayBuilder` method) to get a builder object. You then chain multiple `add` method invocations on the builder object to add the necessary key-value pairs. Finally, the `build` method is invoked to actually return the generated JSON object or JSON array.

The JSON-P object model API provides only getter methods and no setter methods.

JSON-P 1.0 can be used from Java EE 7 (just use any Java EE 7–compatible application server) or from Java SE. To do that in Maven, just make sure you add the following two dependencies in your project object model (POM) file.

```
<dependency>
    <groupId>javax.json</groupId>
    <artifactId>javax.json-api</artifactId>
    <version>1.0</version>
</dependency>

<dependency>
    <groupId>org.glassfish</groupId>
    <artifactId>javax.json</artifactId>
    <version>1.0.4</version>
</dependency>
```

The first `javax.json-api` dependency is needed to compile to code. The second dependency is referencing the JSON-P reference implementation, which is necessary to run JSON-P compiled code.

The following example illustrates how to create a JSON representation of a country using the JSON-P object model's builder API. This example also shows how to handle a very common use case, nesting JSON objects.

```
// 1) get a JSON object builder
JsonObject country = Json.createObjectBuilder()

// 2) add the different key/values pairs
    .add("country", "Belgium")
    . . .
    .add("population", 11200000)
    // note that JSON objects can be nested
    .add("officialLanguages", Json.createArrayBuilder()
         .add(Json.createObjectBuilder()
         .add("language", "Flemish"))
         .add(Json.createObjectBuilder()
         .add("language", "French"))
         .add(Json.createObjectBuilder()
         .add("language", "German")))

// 3) return the generated JSON object
    .build();
```

[For the sake of brevity and clarity, the code snippets omit unrelated but important aspects such as proper error handling, imports, proper resources management, and so forth. —*Ed.*]

The JSON-P object model API provides a variety of getter methods for performing queries on JSON objects (or JSON arrays). Note that the API is immutable and thread-safe. This explains why the API provides only getter methods and no setter methods.

The first parameter passed to a getter is the key of the key-value pair to look up. Optionally, you can pass a second parameter to specify a default value in case that key cannot be found.

```
// looking up the country name value
String capital =
    country.getString("country ",  "Unknown!");
```

Here is an example that uses `JsonReader` to create a JSON object from a file, multiple JSON-P getters to query it, and the Java 8 Stream API to join the results.

```
JsonReader jsonReader =
    Json.createReader(new FileReader("data.json"));
JsonObject country = jsonReader.readObject();

System.out.println("Country: " +
    country.getString("country", "empty!"));

int population = country.getInt("population", 0);
if (population > 0 )
    System.out.println("Population: " + population);

JsonArray langs =
    country.getJsonArray("officialLanguages");

String offLangs =
    langs.getValuesAs(JsonObject.class)
      .stream()
      .map(lang -> lang.getString("language",""))
      .collect(Collectors.joining(", "));

  System.out.println(
      "Official languages: " + offLangs);
```

Using the sample JSON data, this code will produce the following output:

```
Country: Belgium
Population: 11200000
Official languages: Flemish, French, German
```

Using the object model API, you can also navigate through the in-memory object tree. The navigation is based on the common supertype `JsonValue` and is illustrated in the following example. This example shows a simple recursive method, `navigate`, that navigates through the tree structure to display each of its elements. For each tree element, this method invokes the `getValueType` method to get the actual type of the element (for example, a JSON string) so it can then act accordingly (such as invoking the appropriate getter `getString` method).

```
public static void navigate (
    JsonValue tree, String key) {
  if (key != null)
      System.out.print(key + ": ");

  switch(tree.getValueType()) {
    case OBJECT:
      JsonObject object = (JsonObject) tree;
      for (String name : object.keySet())
          navigate (object.get(name), name);
      break;
    case ARRAY:
      System.out.println(" (JSON array)");
      JsonArray array = (JsonArray) tree;
      for (JsonValue val : array)
          navigate (val, null);
      break;
    case STRING:
      JsonString str = (JsonString) tree;
      System.out.println(str.getString());
      break;
    default:
    // for brevity, let's ignore NUMBER,
    //    BOOLEAN and NULL
       break;
  }
}
```

To use this example, just invoke the method and pass it a JSON object:

```
navigate (country, null);
```

The object model API also permits, via the `JsonWriter` class, outputting a JSON object (or array) to a stream. You first use the `Json.createWriter` method to specify the output stream to use. The `JsonWriter.writeObject` method then writes the JSON object to that stream. Finally, you need to close the output stream either by calling the `JsonWriter.close` method or via the `AutoCloseable` "try-with-resources" approach, as illustrated in the example below.

```
StringWriter strWriter = new StringWriter();
try (JsonWriter jsonWriter =
     Json.createWriter(strWriter)) {
    jsonWriter.writeObject(country);
}
```

The JSON-P Streaming API

The second JSON-P API is a lower-level streaming API that is conceptually similar to StAX. This streaming API provides forward-only, read-only access to JSON data in a streaming way. It is particularly well suited for reading, in an efficient manner, large JSON payloads. The streaming API also allows you to write JSON data to output in a streaming fashion.

This API resides in the `javax.json.stream` package. The `JsonParser` interface is at the core of this streaming API. It provides forward-only, read-only access to JSON data using a pull-parsing programming model. In this pull model, the application controls the parser by repeatedly calling `JsonParser` methods to advance the parser. Based on that, the parser state will change, and parser events will be generated to reflect this.

JSON-P obviously is not the first Java-based JSON-related API, but it is the first one that has been standardized through the Java Community Process.

The pull parser can generate any of the following self-explanatory events: `START_OBJECT`, `END_OBJECT`, `START_ARRAY`, `END_ARRAY`, `KEY_NAME`, `VALUE_STRING`, `VALUE_NUMBER`, `VALUE_TRUE`, `VALUE_FALSE`, and `VALUE_NULL`. The application logic should leverage these different events to advance the parser to the necessary position within the JSON document to obtain the required data.

First, create a pull parser using the `Json.createParser` method from either an `InputStream` or a `Reader`. The application will then keep advancing the parser forward by calling the `hasNext` method (Has the parser reached the end yet?) and `next` method on the parser. Keep in mind that the parser can be moved in only one direction: forward.

The following example uses a free online service that exposes country-related information in JSON. The code is simply creating a streaming parser from an `inputStream` using the `Json.createParser` method. The application then keeps advancing the parser to go over each country. In this case, the parsing logic is looking at only two keys: `name` and `capital`. For each `country`, the application looks at the `"name"` value; if it is not `"France"`, the application keeps advancing the parser. Once `"France"` is found, the application looks only at the `"capital"` key. Because the current parser state is `Event.KEY_NAME` (that is, the parser is on `France`'s `capital` key), the application advances the parser one step (`Event.VALUE_STRING`) and gets the actual value of the capital using the `getString` method on the parser. Once this is done, it is useless to continue parsing the rest of the JSON stream, so the application exits the loop.

```
//1. Create a streaming parser from an InputStream
URL url = new URL("http://restcountries.eu/rest/v1/all");
try (InputStream is = url.openStream();
  JsonParser parser = Json.createParser(is)) {

  boolean foundCapital = false;
  boolean foundCountry = false;

//2. Keep advancing the parser
//until it finds the 'France' capital

  while (parser.hasNext() && !foundCapital) {

    Event e = parser.next();
    if (e == Event.KEY_NAME) {

      switch (parser.getString()) {

// is the parser on a pair
// whose key is 'name' and
// value is 'France'?
  case "name":
    parser.next();
    String country = parser.getString();
    if (country.equals("France")) {
        foundCountry = true;
    }
    break;

  case "capital":
    if (foundCountry) {
      // parser is on the 'France' key/value
      // just advance the parser one step
      parser.next();
      // and get the actual value!
      String capital = parser.getString();
      // no need to parse the rest of doc
      foundCapital = true;
      }
    break;

        }
      }
    }
}
```

This example has very simple parsing logic. And depending on the parsing logic requirements, this streaming approach might require you to do a bit more work, but it is clearly more efficient than the higher-level object model–based approach.

Similarly, you can also generate a JSON document in a streaming fashion as illustrated in the example below.

```
FileWriter writer = new FileWriter("canada.json");
JsonGenerator gen = Json.createGenerator(writer);
gen.writeStartObject()
   .write("country", "Canada")
   .write("capital", "Ottawa")
   .write("poulation", 36048521)
   .writeStartArray("officialLanguages")
     .writeStartObject()
       .write("language", "English")
     .writeEnd()
     .writeStartObject()
        .write("language", "French")
     .writeEnd()
   .writeEnd()
   .writeEnd();
gen.close();
```

A `JsonGenerator` is used to write JSON to a byte stream (or to a `Writer`). To obtain a generator, call one of the javax

`.json.Json.createGenerator` static methods. Once you have a `JsonGenerator` instance, you can invoke the different `writeStartObject`, `writeArrayObject`, and `write` methods to construct the representation of the desired JSON object. When you call the `writeStartObject` and `writeArrayObject` methods, it is important to call the corresponding closing method, `writeEnd`. Finally, you need to invoke the `close` method on the generator to properly close resources.

Conclusion

JSON-P provides a simple object model API to parse, generate, and query JSON documents. It also offers an efficient, lower-level API to parse and generate large JSON payloads in a streaming way.

JSON-P is obviously not the first Java-based JSON-related API, but it is the first one that has been standardized through the Java Community Process. And given that JSON-P is now part of Java EE, you can be sure that this API will be available regardless of the Java EE 7 application server you are using. In addition, JSON-P has no dependency on Java EE, so it can also be used in regular Java SE applications. `</article>`

David Delabassée (@delabassee) is a Java veteran and also a regular speaker on the Java conferences circuit. He is currently working at Oracle, where he focuses on server-side Java.

learn more

- "Introducing JSON"
- JSON object model Javadoc
- JSON Stream API Javadoc

Using JavaMail in Java EE

Create a web application that can send emails.

T. LAMINE BA

In this article, I explain how to build a simple web application that uses the core JavaMail API to send email. The application includes three web pages: a front page, a "sent e-mail" confirmation page, and a "failed e-mail" notification page.

The front page of the web application (see **Figure 1**) contains the following web components: input fields for the email address of the sender and recipient, fields for the subject and body, and several fields related to the SMTP server (IP address, username, password, and port number). It also contains the crucial "send" button.

The confirmation page (see **Figure 2**) and the similar notification page for a successful send need only a button that redirects the user back to the front page.

The JavaMail API

The JavaMail API is a package that provides general email facilities, such as reading, composing, and sending electronic messages. JavaMail, which is a platform-independent and protocol-independent framework, is included in Java EE.

As shown in Figure 3, JavaMail has an application-level interface used by the application components to send and receive email. There is also a service provider (SP) interface that speaks protocol-specific languages. For instance, SMTP is used to send emails. Post Office Protocol 3 (POP3) is the standard for receiving emails. Internet Message Access Protocol (IMAP) is an alternative to POP3.

In addition, the JavaMail API contains the JavaBeans Activation Framework (JAF) to handle email content that is not plain text, including Multipurpose Internet Mail Extensions (MIME), URLs, and file attachments.

Required Software

For the purposes of this tutorial, I used the following software: Microsoft Windows (I used Windows 7), the JDK for Java EE 6 or higher, an IDE (I used NetBeans 7), and a web server

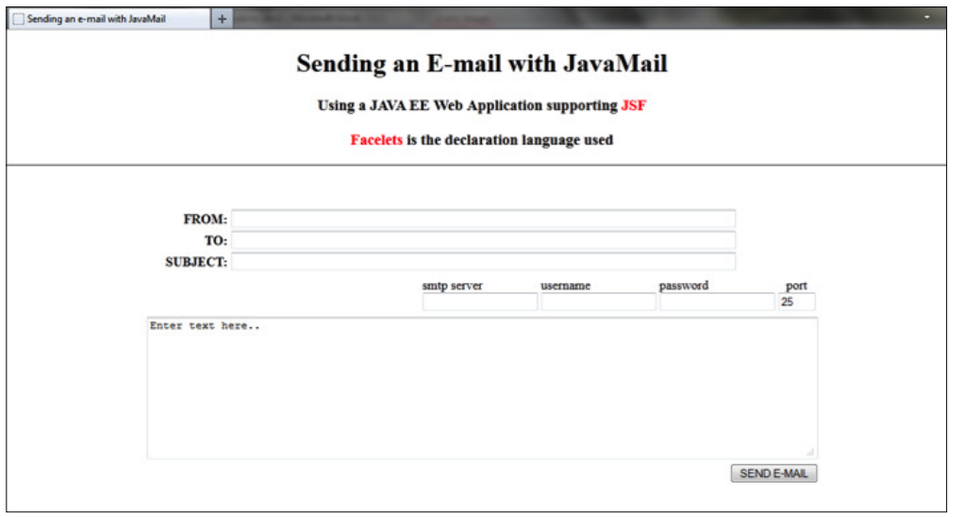


Figure 1. The main page of the email app

Figure 2. The confirmation page

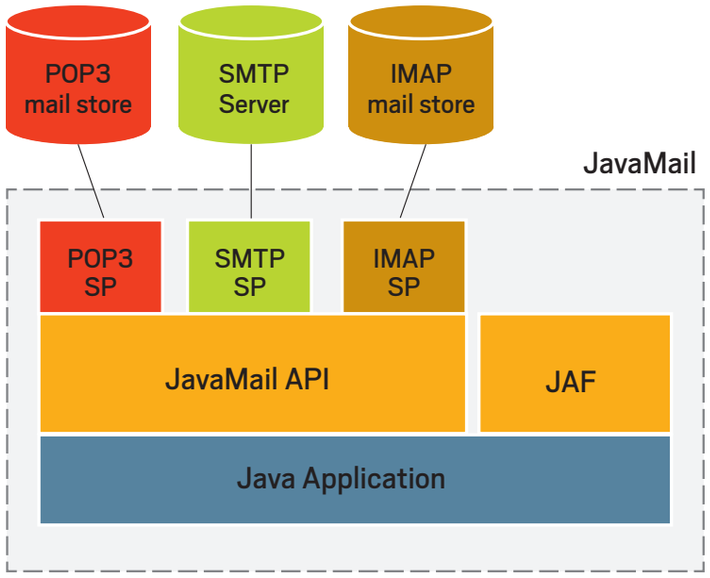


Figure 3. Design of the JavaMail API

such as GlassFish or Apache Tomcat.

Methodology

The tutorial uses JavaServer Faces (JSF) technology to build the web application. Accordingly, the following workflow is proposed:

1. Create a backing bean.
2. Create web pages using component tags.
3. Map the `FacesServlet` instance.

Step 1: Create a Backing Bean

A backing bean is a type of managed bean specific to the JSF technology. It holds the logic of the web application and interacts with the web components contained in the web pages. The backing bean can contain private attributes that correspond to each web component, getter and setter methods referring to the attributes, and methods to handle the following four tasks:

- Perform processing associated with navigation from one web page to another.
- Handle action events.
- Perform validation on a component's value.
- Handle value-change events.

Accordingly, in a backing bean called `emailJSFManagedBean`, we create the getter and setter methods necessary for each of the eight web components listed at the beginning of this article. If the recipient's email address is a variable of type `String` called `to`, **Listing 1** shows how the getter and setter methods would be defined.

Listing 1.

```
package useJavaMail;
/*** Import all necessary libraries ***/

@ManagedBean
@RequestScoped
public class emailJSFManagedBean {
      private String to;
/** Create a new instance of emailJSFManagedBean */
    public emailJSFManagedBean() {
        to = null;
    }

    public String getTo() {
        return to;
    }

    public void setTo(String to) {
        this.to = to;
    }
}
```

In this code, `@ManagedBean` is a declaration that registers the backing bean as a resource with the JSF implementation. In addition, `@RequestScoped` is the annotation that identifies the managed bean as a resource that exists only in the scope of the request. In other words, the bean exists for the duration

of a single HTTP request for the user's interaction with the web application.

We also create two specific methods within the backing bean.

The first method's function is to validate all emails submitted by the user in the web application (see **Listing 2**).

Listing 2.

```
public void validateEmail(FacesContext context,
            UIComponent toValidate, Object value) {

    String message = "";
    String email = (String) value;

    if(email == null || email.equals("")) {
        ((UIInput)toValidate).setValid(false);
        message = "E-mail address is required";
        context.addMessage(
            toValidate.getClientId(context),
            new FacesMessage(message));
    }
    else if (!(email.contains("@") &&
               email.contains("."))) {
        ((UIInput)toValidate).setValid(false);
        message = "E-mail address is invalid";
        context.addMessage(
            toValidate.getClientId(context),
            new FacesMessage(message));
        }
    }
```

Note that the validation email method takes three arguments:

- The context of the JSF implementation, in order to pass error messages from the managed bean to the user interface.
- The identifier `UIComponent toValidate` of the web component that is invoking the method, which, in this case, is a text input field (see **Listing 1**) method, takes user input as an argument, as illustrated in **Figure 1**.
- The variable `value`, which contains the email address that needs to be validated.

Accordingly, the code shown in **Listing 2** accomplishes the following tasks:

- It gets the local value of the web component.
- It checks whether the value is `null` or empty.
- If the value is `null` or empty, the method sets the component's `valid` property to `false` and sets the error message to `E-mail address is required`.
- Otherwise, the method checks whether the `@` character and the period (`.`) character are contained in the value.
- If they aren't, the method sets the component's `valid` property to `false` and sets the error message to `E-mail address is invalid`.

Then, the error message is sent to the `FacesContext` instance, which associates it with the invoking web component.

The second method handles the logic for sending an email with JavaMail. This navigation handling method is triggered by the action of clicking the SEND E-MAIL button (see **Listing 3**).

Listing 3.

```
public String submitEmail() {
// create e-mail and send
    /*** Initialize variables ***/
    props = new Properties();
    // fill props w/ session and message data
    session = Session.getDefaultInstance(
              props, null);
    message = new MimeMessage(session);
    try {
        message.setContent(this.getDescr(),
```

```
                    "text/plain");
message.setSubject(this.getSubject());
fromAddress =
  new InternetAddress(this.getFrom());
message.setFrom(fromAddress);
toAddress =
  new InternetAddress(this.getTo());
message.setRecipient(RecipientType.TO,
                     toAddress);

// Transport message
message.saveChanges(); //send implies save
Transport transport =
  session.getTransport("smtp");
transport.connect(this.smtp, this.port,
                  this.username,
                  this.password);
   if(transport.isConnected() == false)
       return "b_response";
   transport.sendMessage(
       message, message.getAllRecipients());
   transport.close();
}
catch (MessagingException me) {
    // handle catch
    return "b_response";
}

    return "g_response";
}
```

This type of method is known as an *action method*. It is a public method that takes no argument and returns a string that corresponds to the page that the web application will navigate to. In this case, the method produces and sends an email. If the email transmission is successful, the method returns `g_response` (for "good response"), which displays the page g_response.xhtml in the browser (see Figure 2). If the email transmission fails, `b_response` (for "bad response") is returned and the browser displays the page b_response.xhtml.

To send an email using JavaMail, we first initiate an email session instance with the `Session` class. The email session is the starting point for JavaMail. It uses the `java.util.Properties` class to get information, such as the email server, the username, and the password, which can be shared across the rest of the application. In this case, we create a default instance of the `Session` class:

```
session =
    Session.getDefaultInstance(props, null);
```

Second, through the session, we produce the email using the `Message` class. However, considering that `Message` is an abstract class, we choose instead its subclass `MimeMessage`, which allows us to create messages that understand MIME types and headers, as defined in the different standards-defining RFCs. The message is constructed with the session as an argument:

```
MimeMessage message =
    new MimeMessage(session)
```

Then, we send the email by manipulating an object of type `Transport`. The message is sent via the transport protocol SMTP. The transmission is handled by the `Transport` class and an object is instantiated as follows:

```
Transport transport =
    session.getTransport("smtp");
```

Then, the transport object attempts to connect to the SMTP

server using the suggested credentials (the SMTP server address, the port number that accepts SMTP connections, the username, and the password) to pass authentication on the server.

```
transport.connect(this.smtp,
    this.port, this.username,
    this.password);
```

If the connection is accepted by the SMTP server, the email is sent via the `send` command.

Finally, we close the transportation service by invoking the `close` command:

```
transport. sendMessage(message,
    message.getAllRecipients());
    transport.close();
```

Note that the file containing the backing bean should be under the Sources Packages directory of the web application.

Step 2: Create Web Pages Using Component Tags

The different web pages of the application take advantage of the Facelets declaration language to produce tags for various web components.

Create the front page. On this page (**Figure 1**), there are four types of tags associated with the web components: `inputText`, `inputSecret`, `inputTextArea`, and `commandButton`. The `inputText` is equivalent to an input tag of type `text` in HTML. In other words, it is a field that takes user input. We use this type of tag to obtain the sender's address, the recipient's address, the subject of the email, the SMTP server address, the SMTP server username, and the port number of the SMTP server.

The `inputSecret` is equivalent to the input tag of type `password` in HTML. It is also a field that takes user input. However, contrary to the `inputText` tag, the `inputSecret` does not display the value entered by the user. This tag is used to record the SMTP server password.

The `inputTextArea` is equivalent to the `textarea` tag in HTML. It is used to record the body of the email we intend to send.

User input is validated by the application either by using the standard validators or by invoking a validating method implemented in the backing bean (see **Listing 2**).

For example, using Facelets, we invoke the validating method `emailJSFManagedBean.validateEmail` for the FROM address field using the code shown in **Listing 4**.

Listing 4.

```
<h:form>
 <table>
  <tr>
   <th style="width:100px"
     align="right">FROM:</th>
   <td>
    <h:inputText id="from" size="100"
     validator=
       "#{emailJSFManagedBean.validateEmail}"
     value="#{emailJSFManagedBean.from}" />
     <span style="margin-left:10px">
       <h:message style="color:red" for="from"/>
     </span>
   </td>
  </tr>
 </table>
</form>
```

Note that the purpose of the message tag (`<h:message/>`) is to display the error message if the email address validation fails.

As another example, **Listing 5** shows how we use standard validators in Facelets for the SUBJECT field.

Listing 5.

```
<h:form>
 <table>
  <tr>
   <th style="width:100px" align="right">FROM:</th>
    <td>
     <h:inputText id="subject" size="100"
       validatorMessage="Subject is required"
       value="#{emailJSFManagedBean.subject}">
     <f:validateRequired  for="subject"/>
     </h:inputText>
     <span style="margin-left:10px">
       <h:message style="color:red" for="from"/>
     </span>
    </td>
   </tr>
 </table>
</form>
```

The validateRequired tag (<f:validateRequired/>) is applied to the inputText with an id of subject. This invalidates the form when it is submitted and the SUBJECT field is empty. In this case, an error message is displayed where the message tag (<h:message/>) is located.

Create the confirmation page and the error notification page. The confirmation page (g_response.xhtml) is called when an email has been sent (see Figure 2). On the other hand, the error notification page (b_response.xhtml) is called when the email transmission fails. Both pages contain, respectively, only one web component, which is a Facelets commandButton tag:

```
<h:form>
  <h:commandButton id="back"
     value="Back" action="index">
</h:form>
```

The code creates a button that, when clicked, moves the user to the front page (index.xhtml).

Step 3: Map the FacesServlet Instance

The final step consists of mapping the FacesServlet instance by altering the web deployment descriptor, that is, the web .xml file. Listing 6 is a typical example.

Listing 6.

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>
    javax.faces.webapp.FacesServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```

Note, however, that the mapping is done automatically if you are using an IDE such as NetBeans.

These examples have demonstrated how to use only the basic functionality of the JavaMail API. The library, however, is much more extensive and covers almost all the needs of a mail agent. While JavaMail was designed for use with Java EE, it can be used with Java SE, which can make for fun projects.

[This article, like others in the *//from the vault /* series, is an updated version of an article that appeared in an earlier issue of *Java Magazine*. This article first appeared in the March/April 2012 issue. —*Ed.*] </article>

T. Lamine Ba is the president of Real Basis and cofounder of the West African Java user group SeneJUG and is one of the early members of JUG-Africa.

CONSTANTIN DRABO

JShell: Read-Evaluate-Print Loop for the Java Platform

Testing code snippets will be part of the JDK.

JShell, a new read-evaluate-print loop (REPL), will be introduced in JDK 9. Motivated by Project Kulla (JEP 222), JShell is intended to provide developers an API and an interactive tool that evaluates declarations, statements, and expressions of the Java programming language.

In this article, I present a brief overview of JShell, explain its use, and demonstrate its benefits for developers.

Overview

JShell is a new tool in JDK 9 that offers a basic shell for Java that uses a command-line interface. It is also the first official REPL implementation for the Java platform, although this concept has existed in many languages (for example, Groovy and Lisp) and in third-party tools (such as Java REPL and BeanShell).

JShell acts like a UNIX shell: it reads the instructions, evaluates them, prints the result of the instructions, and then displays a prompt while waiting for new commands. It is built around several core concepts—snippets, state, wrapping, instruction modification, forward references, and snippet dependencies—that I explain.

A *snippet* corresponds to an instruction that is based on Java Language Specification (JLS) syntax. It represents a single expression, statement, or declaration. What follows is a simple snippet. When you enter the snippet into JShell, the line below is displayed by the REPL:

```
System.out.println("My JShell snippet");
My JShell snippet
```

In my examples in this article, the characters in blue indicate text entered at the command line into JShell, and the resulting output is shown in black monospace font.

Like Java code, JShell allows you to declare variables, methods, and classes:

```
int x, y , sum
|  Added variable x of type int
|  Added variable y of type int
|  Added variable sum of type int

x = 10 ; y = 20 ; sum = x + y;
|  Variable x has been assigned the value 10
|  Variable y has been assigned the value 20
Variable sum has been assigned the value 30

System.out.println("Sum of " + x + " and " + y +
  " = " + sum);
Sum of 10 and 20 = 30
```

And now, here's an example of a valid class, which I use later:

```
class Student {
private String name ;
private String classRoom ;
private double grade ;

public Student() {

}

public String getName() {
return name ;
}

public void setName(String name) {
this.name =  name ;
}

public String getClassRoom() {
return classRoom ;
}

public void setClassRoom(String classRoom) {
this.classRoom  = classRoom ;
}

public double getGrade() {
return grade ;
}

public void setGrade(double grade) {

this.grade  = grade ;
}
}
|  Added class Student
```

The indentation, of course, looks different than in Java, because this code was typed at the JShell command line.

Note that some normal Java statements are not allowed at this initial declaration. The only permitted class modifier is `abstract`. Packages are not allowed. Even `public` won't work:

```
public class University {
Student student  = new Student();
}
|  Warning:
|  Modifier 'public' not permitted in top-level
   declarations, ignored
|  public class University {
|  ^----^
|  Added class University
```

State. Each statement in JShell has a state. The state defines the execution status of snippets and of variables. It is determined by results of the `eval()` method of the JShell instance, which evaluates code. There are seven status states:

- DROPPED: The snippet is inactive.
- NONEXISTENT: The snippet is inactive because it does not yet exist.
- OVERWRITTEN: The snippet is inactive because it has been replaced by a new snippet.
- RECOVERABLE_DEFINED: The snippet is a declaration snippet with potentially recoverable unresolved references or other issues in its body.
- RECOVERABLE_NOT_DEFINED: The snippet is a declaration snippet with potentially recoverable unresolved references or other issues. (I discuss the difference between this and the previous state shortly.)
- REJECTED: The snippet is inactive because it failed compilation upon initial evaluation and it is not capable of becoming valid with further changes to the JShell state.
- VALID: The snippet is valid.

When a snippet is not declared, it is considered inactive and not part of the state of the JShell instance nor is it visible to the compilation of other snippets. At this stage, it is a NONEXISTENT snippet.

If the snippet is submitted to the `eval()` method and there are no errors, it becomes part of the state of the JShell instance and the status is VALID. Querying JShell gives `isDefined == true` and `isActive == true`.

In the case where the signature of the snippet is valid but the body contains issues or unresolved references, the status is RECOVERABLE_DEFINED and a JShell query states `isDefined == true` and `isActive == true`.

If the signature of the snippet is wrong and the body also contains issues or unresolved references, the snippet's status is RECOVERABLE_NOT_DEFINED and the status is `isDefined == false` even though the snippet stays active (`isActive == true`).

A snippet becomes REJECTED when compilation fails, and it is no longer a valid snippet. This is a final status and will not change again. At this stage, both `isDefined` and `isActive` are set to `false`.

You can also deactivate and remove a snippet from the JShell state with an explicit call to the `JShell.drop(jdk.jshell.PersistentSnippet)` method. At that point, the snippet status changes to DROPPED. This is also a final status and will not change in the future.

Sometimes a snippet type declaration matches another one. In this case, the previous snippet is inactive and it is replaced by the new one. The status of the old snippet becomes OVERWRITTEN and the snippet is no longer visible to other snippets (`isActive == false`). OVERWRITTEN is also a final status.

Using JShell from a Program

OpenJDK offers APIs to developers access to JShell programmatically rather than by using the REPL. The following code creates an instance of JShell, evaluates a snippet, and provides the status of the instructions.

```
import java.util.List;
import jdk.jshell.*;
import jdk.jshell.Snippet.Status;

public class JShellStatusSample {
  public static void main(String... args) {

    //Create a JShell instance
    JShell shell = JShell.create();

    //Evaluate the Java code
    List<SnippetEvent> events =
      shell.eval( "int x, y, sum; " +
        "x = 15; y = 23; sum = x + y; " +
        "System.out.println(sum)" );
    for(SnippetEvent event : events) {
      //Create a snippet instance
      Snippet snippet = event.snippet();
      //Store the status of the snippet
      Snippet.Status snippetstatus =
        shell.status(snippet);
      if(snippetstatus == Status.VALID) {
          System.out.println("Successful");
      }
    }
}
```

The result of the execution of this code is

```
java JShellStatusSample
Successful
Successful
Successful
```

Wrapping

You are not obliged to declare variables or define a method within a class. Classes, variables, methods, expressions, and statements evolve within a synthetic class (as an artificial block). You can define them in the top-level context or within a class body, as you wish.

```
String firstName , lastName ;
| Added variable firstName of type String
| Added variable lastName of type String

String concatName(String firstName,
String lastName) {
return firstName + lastName ;
}
|  Added method concatName(String,String)
```

The following code shows the declaration of variables and a method in the top-level context. As discussed previously, you cannot modify classes at the top level; however, as seen in the following code, you can modify methods within classes.

```
class Person {

private String firstName ;
private String lastName ;

public String concatName(String firstName,
String lastName) {
return firstName + lastName;
}

}
|  Added class Person
```

Because each statement or expression is created in its own unique namespace, modifications can be applied at any time without disturbing the overall functioning of the code.

Forward References and Dependencies

Within the body of a class, you can refer to members that will be defined later. During evaluation of the code, the references produce errors. But because JShell works sequentially, the issue can be resolved by writing the missing member before actually calling the snippets.

When a snippet A depends on a second snippet B, any changes in snippet B are immediately propagated in A. Then, if the dependent snippet is updated, the main snippet is also updated. If the dependent snippet is invalid, the main snippet becomes invalid.

How to Run JShell

To run JShell, you need to download and install the latest early-access preview build for JDK 9 for your environment. Then, set your JAVA_HOME environment variable and run `java -version` to verify your installation. The output of the command should show something like the following:

```
java version "9-ea"
Java(TM) SE Runtime Environment (build 9-ea+100-2016-
  01-06-195905.javare.4235.nc)
Java HotSpot(TM) 64-Bit Server VM
...
```

To run JShell, type `jshell` at the command line:

```
[pandaconstantin@localhost ~]$ jshell
|  Welcome to JShell -- Version 9-ea
|  Type /help for help
```

When the prompt is available, you can get help on several useful commands by typing `/help` at the command line.

Figure 1 shows the truncated output from that command.

If you declare variables and then initialize them, you can see them by using the `list` command, for example:

```
String firstname;
|  Added variable firstname of type String

String lastname;
|  Added variable lastname of type String

double grade;
|  Added variable grade of type double

String getStudentFullName(String firstname,
        String lastname) {
return firstname + " " + lastname ; }
| Added method getStudentFullName(String,String)

firstname  =  "Wolfgang" ;
|  Variable firstname has been assigned the
 value "Wolfgang"

lastname  = "Mozart";
|  Variable lastname has been assigned the
 value "Mozart"

System.out.println("Hello " +
getStudentFullName(firstname,lastname));
Hello Wolfgang Mozart
```

The output of the `list` command shows the following:

```
1 : String firstname ;
2 : String lastname ;
3 : double grade ;
```

```
/list [all|start|history|<name or id>] -- list the source you have typed
/seteditor <executable>                -- set the external editor command to use
/edit <name or id>                     -- edit a source entry referenced by name or id
/drop <name or id>                     -- delete a source entry referenced by name or id
/save [all|history|start] <file>       -- save: <none> - current source;
                                               all - source including overwritten, failed,
                                                     and start-up code;
                                               history - editing history;
                                               start - default start-up definitions
/open <file>                           -- open a file as source input
/vars                                  -- list the declared variables and their values
/methods                               -- list the declared methods and their signatures
/classes                               -- list the declared classes
/imports                               -- list the imported items
/exit                                  -- exit the REPL
/reset                                 -- reset everything in the REPL
/feedback <level>                      -- feedback information: off, concise, normal, verbose,
                                              default, or ?
```

Figure 1. Partial list of JShell commands

```
4 : String getStudentFullName
(String firstname, String lastname) {
     return firstname + " " + lastname ;
 }
5 : firstname = "Wolfgang" ;
6 : lastname  = "Mozart" ;
7 : System.out.println("Hello " +
getStudentFullName(firstname,lastname));
```

The numbers in the output are the snippet identifiers. They are useful for manipulating a snippet (editing, dropping, and so on.) You can also list all the variables, methods, and classes that are in the code. Here's an example of listing all the variables:

```
/vars
|    String firstname = "Wolfgang"
|    String lastname = "Mozart"
|    double grade = 0.0
```

If you decide to change the values of variables or edit a specific snippet, you run `/edit` with the snippet identifier, for example:

```
/edit 5
```

A dialog box appears, which allows you to modify the value. After you make the change in the dialog box, you will see output that looks like this:

```
|  Variable firstname has been assigned
 the value "Constantin"
```

Here's another example:

```
/edit 6
|  Variable lastname has been assigned
 the value "Drabo"
```

When I rerun snippet 7, the output is updated accordingly:

```
/7
System.out.println("Hello " +
getStudentFullName(firstname, lastname));
Hello Constantin Drabo
```

The `/save` command enables you to save your snippets to a file, and the `/open` command enables you to open and run the file:

```
/save StudentName.jsh
/open StudentName.jsh
```

JShell also offers some keyboard shortcuts. You can obtain the navigation history by using the up and down arrow keys or the Enter key. Use the tab key to perform snippet completion, and interrupt a snippet by using Control-C.

Conclusion

JShell has many possible uses: for testing code, especially APIs; for educational purposes; and for doing quick mock-ups in JavaFX.

Whether it is called from the command line or programmatically, JShell is likely to become one of the most widely used features of JDK 9. `</article>`

Constantin Drabo is a software engineer living in Burkina Faso. He is a NetBeans Dream Teamer and a Fedora Ambassador for the Fedora Project. He is also the founder of FasoJUG, the first Java user group in Burkina Faso (the former Upper Volta).

BENJAMIN **EVANS** AND
DAVID **FLANAGAN**

Modern Java I/O

NIO.2 makes many things easier, including monitoring directories for changes.

Java 7 brought in a brand-new I/O API—usually called NIO.2—and it should be considered almost a complete replacement for the original `File` approach to I/O. The new classes are contained in the `java.nio.file` package.

The new API is considerably easier to use for many use cases. It has two major parts. The first is a new abstraction called `Path` (which can be thought of as representing a file location, which may or may not have anything actually at that location). The second piece is lots of new convenience and utility methods to deal with files and filesystems. These are contained as static methods in the `Files` class.

For example, when using the new `Files` functionality, a basic copy operation is now as simple as

```
File inputFile = new File("input.txt");
try (InputStream in =
       new FileInputStream(inputFile)) {
  Files.copy(in, Paths.get("output.txt"));
} catch(IOException ex) {
  ex.printStackTrace();
}
```

Let's take a quick survey of some of the major methods in `Files`—the operation of most of them is pretty self-explanatory. In many cases, the methods have return types. We have omitted handling these, as they are rarely useful except for contrived examples, and for duplicating the behavior of the equivalent C code:

```
Path source, target;
Attributes attr;
Charset cs = StandardCharsets.UTF_8;

// Creating files
//
// Example of path --> /home/ben/.profile
// Example of attributes --> rw-rw-rw-
Files.createFile(target, attr);

// Deleting files
Files.delete(target);
boolean deleted = Files.deleteIfExists(target);

// Copying/Moving files
Files.copy(source, target);
Files.move(source, target);

// Utility methods to retrieve information
long size = Files.size(target);

FileTime fTime =
     Files.getLastModifiedTime(target);
System.out.println(fTime.to(TimeUnit.SECONDS));

Map<String, ?> attrs =
    Files.readAttributes(target, "*");
System.out.println(attrs);
```

BEN EVANS PHOTOGRAPH BY JOHN BLYTHE

```
// Methods to deal with file types
boolean isDir = Files.isDirectory(target);
boolean isSym = Files.isSymbolicLink(target);

// Methods to deal with reading and writing
List<String> lines =
    Files.readAllLines(target, cs);
byte[] b = Files.readAllBytes(target);

BufferedReader br =
    Files.newBufferedReader(target, cs);
BufferedWriter bwr =
    Files.newBufferedWriter(target, cs);

InputStream is = Files.newInputStream(target);
OutputStream os = Files.newOutputStream(target);
```

Some of the methods on `Files` provide the opportunity to pass optional arguments, to provide additional (possibly implementation-specific) behavior for the operation.

Some of the API choices here produce occasionally annoying behavior. For example, by default, a copy operation will not overwrite an existing file, so we need to specify this behavior as a copy option:

```
Files.copy(Paths.get("input.txt"),
           Paths.get("output.txt"),
           StandardCopyOption.REPLACE_EXISTING);
```

`StandardCopyOption` is an enum that implements an interface called `CopyOption`. This is also implemented by `LinkOption`. So `Files.copy()` can take any number of either `LinkOption` or `StandardCopyOption` arguments. `LinkOption` is used to specify how symbolic links should be handled (provided the underlying operating system supports symlinks, of course).

Path

Path is a type that may be used to locate a file in a filesystem. It represents a path that is

- System-dependent
- Hierarchical
- Composed of a sequence of path elements
- Hypothetical (may not exist yet, or may have been deleted)

It is therefore fundamentally different from a `File`. In particular, the system dependency is manifested by `Path` being an interface, not a class. This enables different filesystem providers to each implement the `Path` interface and provide for system-specific features while retaining the overall abstraction.

Path is an abstract concept; it isn't necessarily bound to any physical file path, so you can talk easily about the locations of files that don't exist yet.

The elements of a `Path` consist of an optional root component, which identifies the filesystem hierarchy that this instance belongs to. Note that, for example, relative `Path` instances may not have a root component. In addition to the root, all `Path` instances have zero or more directory names and a name element.

The name element is the element farthest from the root of the directory hierarchy and represents the name of the file or directory. The `Path` can be thought of consisting of the path elements joined together by a special separator or delimiter.

`Path` is an abstract concept; it isn't necessarily bound to any physical file path. This allows us to talk easily about the locations of files that don't exist yet. Java ships with a `Paths` class that provides factory methods for creating `Path` instances.

`Paths` provides two `get()` methods for creating `Path` objects. The usual version takes a `String` and uses the default filesystem provider. The URI version takes advantage of the ability of NIO.2 to plug in additional providers of bespoke filesystems. This is an advanced usage, and interested developers

should consult the primary documentation:

```
Path p = Paths.get("/Users/ben/cluster.txt");
Path p =
  Paths.get(new URI(
            "file:///Users/ben/cluster.txt"));
System.out.println(p2.equals(p));

File f = p.toFile();
System.out.println(f.isDirectory());
Path p3 = f.toPath();
System.out.println(p3.equals(p));
```

This example also shows the easy interoperation between `Path` and `File` objects. The addition of a `toFile()` method to `Path` and a `toPath()` method to `File` allows the developer to move effortlessly between the two APIs and allows for a straightforward approach to refactoring the internals of code based on `File` to use `Path` instead.

We can also make use of some useful "bridge" methods that the `Files` class also provides. These provide convenient access to the older I/O APIs—for example, by providing convenience methods to open `Writer` objects to specified `Path` locations:

```
Path logFile = Paths.get("/tmp/app.log");
try (BufferedWriter writer =
       Files.newBufferedWriter(
           logFile,
           StandardCharsets.UTF_8,
           StandardOpenOption.WRITE)) {
  writer.write("Hello World!");
  // ...
} catch (IOException e) {
  // ...
}
```

We're making use of the `StandardOpenOption` enum, which provides similar capabilities to the copy options but for opening a new file instead.

In the next example, we manipulate a JAR file as a `FileSystem` in its own right, modifying it to add an additional file directly into the JAR. JAR files are just ZIP files, so this technique will also work for .zip archives:

```
Path tempJar = Paths.get("sample.jar");
try (FileSystem workingFS =
  FileSystems.newFileSystem(tempJar, null)) {
  Path pathForFile =
      workingFS.getPath("/hello.txt");
  List<String> ls = new ArrayList<>();
  ls.add("Hello World!");

  Files.write(pathForFile, ls,
              Charset.defaultCharset(),
              StandardOpenOption.WRITE,
              StandardOpenOption.CREATE);
}
```

This shows how we use a `FileSystem` to make the `Path` objects inside it, via the `getPath` method. This enables the developer to effectively treat `FileSystem` objects as black boxes.

One of the criticisms of Java's original I/O APIs was the lack of support for native and high-performance I/O. A solution was initially added in Java 1.4, the Java New I/O (NIO) API, and it has been successively refined in successive Java versions.

NIO Channels and Buffers

NIO buffers are a low-level abstraction for high-performance I/O. They provide a container for a linear sequence of elements of a specific primitive type. We'll work with the `ByteBuffer` (the most common case) in our examples.

ByteBuffer. This is a sequence of bytes, and can conceptually be thought of as a performance-critical alternative to working with a `byte[]`. To get the best possible performance, `ByteBuffer` provides support for dealing directly with the native capabilities of the platform the JVM is running on.

This approach is called the "direct buffers" case, and it bypasses the Java heap wherever possible. Direct buffers are allocated in native memory, not on the standard Java heap, and they are not subject to garbage collection in the same way as regular on-heap Java objects.

To obtain a direct `ByteBuffer`, call the `allocateDirect()` factory method. An on-heap version, `allocate()`, is also provided, but in practice this is not often used.

A third way to obtain a byte buffer is to wrap an existing `byte[]`—this will give an on-heap buffer that serves to provide a more object-oriented view of the underlying bytes:

```
ByteBuffer b = ByteBuffer.allocateDirect(65536);
ByteBuffer b2 = ByteBuffer.allocate(4096);

byte[] data = {1, 2, 3};
ByteBuffer b3 = ByteBuffer.wrap(data);
```

Byte buffers are all about low-level access to the bytes. This means that developers have to deal with the details manually—including the need to handle the endianness of the bytes and the signed nature of Java's integral primitives:

```
b.order(ByteOrder.BIG_ENDIAN);

int capacity = b.capacity();
int position = b.position();
int limit = b.limit();
int remaining = b.remaining();
boolean more = b.hasRemaining();
```

To get data in or out of a buffer, we have two types of operation—single value, which reads or writes a single value, and bulk, which takes a `byte[]` or `ByteBuffer` and operates on a (potentially large) number of values as a single operation. It is from the bulk operations that performance gains would expect to be realized:

```
b.put((byte)42);
b.putChar('x');
b.putInt(0xcafebabe);

b.put(data);
b.put(b2);

double d = b.getDouble();
b.get(data, 0, data.length);
```

The single value form also supports a form used for absolute positioning within the buffer:

```
b.put(0, (byte)9);
```

Buffers are an in-memory abstraction. To affect the outside world (for example, the file or network), we need to use a `Channel`, from the package `java.nio.channels`. Channels represent connections to entities that can support read or write operations. Files and sockets are the usual examples of channels, but we could consider custom implementations used for low-latency data processing.

Channels are open when they're created, and can subsequently be closed. Once closed, they cannot be reopened. Channels are usually either readable or writable, but not both. The key to understanding channels is that reading from a channel puts bytes into a buffer, and writing to a channel takes bytes from a buffer. For example, suppose we have a large file that we want to checksum in 16 MB chunks:

```
FileInputStream fis = getSomeStream();
boolean fileOK = true;

try (FileChannel fchan = fis.getChannel()) {
  ByteBuffer buffy =
      ByteBuffer.allocateDirect(16 * 1024 * 1024);
  while(fchan.read(buffy) != -1 ||
        buffy.position() > 0 ||
        fileOK) {
    fileOK = computeChecksum(buffy);
    buffy.compact();
  }
} catch (IOException e) {
  System.out.println("Exception in I/O");
}
```

This will use native I/O as far as possible, and will avoid a lot of copying of bytes on and off the Java heap. If the `computeChecksum` method has been well implemented, then this could be a very performant implementation.

Mapped byte buffers. These are a type of direct byte buffer that contain a memory-mapped file (or a region of one). They are created from a `FileChannel` object, but note that the `File` object corresponding to the `MappedByteBuffer` must not be used after the memory-mapped operations, or an exception will be thrown. To mitigate this, we again use try-with-resources, to scope the objects tightly:

```
try (RandomAccessFile raf =
  new RandomAccessFile(
      new File("input.txt"), "rw");
      FileChannel fc = raf.getChannel();) {

  MappedByteBuffer mbf =
    fc.map(FileChannel.MapMode.READ_WRITE, 0,
          fc.size());
  byte[] b = new byte[(int)fc.size()];
  mbf.get(b, 0, b.length);
  for (int i=0; i<fc.size(); i++) {
    b[i] = 0; // Won't be written back to the
              // file, we're a copy
  }
  mbf.position(0);
  mbf.put(b); // Zeroes the file
}
```

Even with buffers, there are limitations to what can be done in Java for large (for example, transferring 10 GB between filesystems) I/O operations that perform synchronously on a single thread. Before Java 7, these types of operations would typically be done by writing custom multithreaded code and managing a separate thread for performing a background copy. Let's move on to look at the new asynchronous I/O features that were added with JDK 7.

Async I/O

The key to the new asynchronous functionality is some new subclasses of `Channel` that can deal with I/O operations that need to be handed off to a background thread. The same functionality can be applied to large, long-running operations, and to several other use cases.

In this section, we'll deal exclusively with `Asynchronous FileChannel` for file I/O, but there are a couple of other asynchronous channels. There are two different ways to interact with an asynchronous channel—Future style, and callback style.

Future-based style. The Future interface is beyond the scope of this article, but it can be thought of as an ongoing task that may or may not have completed yet. It has two key methods: `isDone()`, which returns a Boolean indicating whether the task has finished, and `get()`, which returns the result. If the task is finished, it returns immediately. If not finished, it

blocks until done.

Here is an example of a program that reads a large file (possibly as large as 100 MB) asynchronously:

```
try (AsynchronousFileChannel channel =
         AsynchronousFileChannel.open(
             Paths.get("input.txt"))) {
  ByteBuffer buffer =
      ByteBuffer.allocateDirect(1024 * 1024 * 100);
  Future<Integer> result = channel.read(buffer, 0);

  while(!result.isDone()) {
    // Do some other useful work....
  }

  System.out.println("Bytes read: " + result.get());
}
```

Callback-based style. The callback style for asynchronous I/O is based on a `CompletionHandler`, which defines two methods, `completed()` and `failed()`, that will be called back when the operation either succeeds or fails.

This style is useful if you want immediate notification of events in asynchronous I/O—for example, if there are a large number of I/O operations in flight, but failure of any single operation is not necessarily fatal:

```
byte[] data = {2, 3, 5, 7, 11, 13, 17, 19, 23};
ByteBuffer buffy = ByteBuffer.wrap(data);

CompletionHandler<Integer,Object> h =
  new CompletionHandler() {
  public void completed(Integer written, Object o) {
    System.out.println("Bytes written: " + written);
  }
  public void failed(Throwable x, Object o) {
    System.out.println(
        "Async write failed: "+ x.getMessage());
  }
};

try (AsynchronousFileChannel channel =
       AsynchronousFileChannel.open(
          Paths.get("primes.txt"),
          StandardOpenOption.CREATE,
          StandardOpenOption.WRITE)) {

  channel.write(buffy, 0, null, h);
  Thread.sleep(1000); // So we don't exit too quickly
}
```

The `AsynchronousFileChannel` object is associated with a background thread pool, so that the I/O operation proceeds, while the original thread can get on with other tasks.

By default, this uses a managed thread pool that is provided by the runtime. If required, it can be created to use a thread pool that is managed by the application (via an overloaded form of `AsynchronousFileChannel.open()`), but this is not often necessary.

Finally, for completeness, let's touch upon NIO's support for multiplexed I/O. This enables a single thread to manage multiple channels and to examine those channels to see which are ready for reading or writing. The classes to support this are in the `java.nio.channels` package and include `SelectableChannel` and `Selector`.

These nonblocking multiplexed techniques can be extremely useful when writing advanced applications that require high scalability, but a full discussion is outside the scope of this article.

Watch Services and Directory Searching

The last class of asynchronous services we will consider watch a directory or visit a directory (or a tree). The watch services operate by observing everything that happens in a directory—for example, the creation or modification of files:

```
try {
  WatchService watcher =
      FileSystems.getDefault().newWatchService();

  Path dir =
      FileSystems.getDefault().getPath("/home/ben");
  WatchKey key =
      dir.register(watcher,
                   StandardWatchEventKinds.ENTRY_CREATE,
                   StandardWatchEventKinds.ENTRY_MODIFY,
                   StandardWatchEventKinds.ENTRY_DELETE);

  while(!shutdown) {
    key = watcher.take();
    for (WatchEvent<?> event: key.pollEvents()) {
      Object o = event.context();
      if (o instanceof Path) {
        System.out.println("Path altered: "+ o);
      }
    }
    key.reset();
  }
}
```

By contrast, the directory streams provide a view into all files currently in a single directory. For example, to list all the Java source files and their size in bytes, we can use code like

```
try(DirectoryStream<Path> stream =
    Files.newDirectoryStream(
        Paths.get("/opt/projects"), "*.java")) {
  for (Path p : stream) {
    System.out.println(p +": "+ Files.size(p));
  }
}
```

One drawback of this API is that this will only return elements that match according to glob syntax, which is sometimes insufficiently flexible. We can go further by using the new `Files.find` and `Files.walk` methods to address each element obtained by a recursive walk through the directory:

```
final Pattern isJava = Pattern.compile(".*\\.java$");
final Path homeDir = Paths.get("/Users/ben/projects/");
Files.find(homeDir, 255,
  (p, attrs) -> isJava.matcher(p.toString()).find())
     .forEach(
       q -> {System.out.println(q.normalize());});
```

It is possible to go even further and construct advanced solutions based on the `FileVisitor` interface in `java.nio.file`, but that requires the developer to implement all four methods on the interface rather than just using a single lambda expression as done here.

In sum, you can see that the NIO.2 library provides a lot of useful functionality and saves you a lot of code. If you're still working with pre–Java 7 file handling, you're doing far more work than necessary. </article>

This article was adapted with permission from Java in a Nutshell, *by Benjamin Evans and David Flanagan.*

Benjamin Evans is the cofounder of jClarity, a Java Champion and Rock Star, and a frequent contributor to *Java Magazine*. **David Flanagan** is a software engineer at Mozilla, best known for his master work *JavaScript: the Definitive Guide* (O'Reilly, 2011).

MICHAEL KÖLLING

Generics: The Hard Parts

Wildcards, subtyping, and type erasure in generics

Welcome back to the discussion of generic types in Java. In my previous article, I started discussing generics types—why they are useful, what you can do with them, and how to use them. The introductory part of this topic was quite straightforward, but at the end of that discussion I mentioned a problem: generic collections and subtyping.

In short, I wanted to write a general `printList` method such as this:

```
private void printList(List<Person> list)
```

And I wanted it to print out lists of subtypes of `Person`, such as `List<Student>` or `List<Faculty>`. In other words, given that `Student` is a subtype of `Person`, I wanted to call the method above like this:

```
List<Student> students = getStudentList();
printList(students);
```

This does not work in Java. The reason is that `List<Student>` is not considered a subtype of `List<Person>` even though `Student` is a subtype of `Person`.

What's the Problem?

So why is `List<Student>` not a subtype of `List<Person>`? If you think only about printing out the list, there seems to be no problem. The `printList` method could call, for instance, a `print` method on all the list's elements (which might be defined in `Person` and redefined appropriately in the subtypes). All seems well.

The problem becomes apparent when you consider that the `printList` method could also modify the list. It could, for example, include the following line:

```
list.add(new Faculty());
```

Because the static type of the `list` variable (the formal parameter to the method) is `List<Person>`, and `Faculty` is a subtype of `Person`, adding this object causes no type problems. However, if the actual list passed to the `printList` method were a list of students, then I have now added a `Faculty` object to the `Student` list! This is a clear error and should not be allowed to happen.

The only solution is to declare that `List<Student>` is not a subtype of `List<Person>`, and to prevent student lists from being passed in to the `printList` method. Type safety is preserved, but I am back to square one: How can I now write my general `printList` method?

Wildcards to the Rescue

The solution to this problem is the use of wildcards. I can write my `printList` method like this:

```
private void printList(List<?> list)
```

Note the question mark in place of the element type of

PHOTOGRAPH BY JOHN BLYTHE

the list. The question mark is the wildcard symbol, and it denotes a type called *unknown*. My parameter is now a *list of unknown type*.

There is an obvious benefit to this construct. I can now do what I intended to do: I can call my `printList` method with both `List<Student>` and `List<Faculty>` as parameters:

```
List<Student> students = getStudentList();
List<Faculty> professors = getFacultyList();
printList(students);
printList(professors);
```

Every list type is considered to be a subtype of the list of this unknown type, so this code now works. The trade-off is that I cannot add to the list when the element type is unknown, so I avoid the type problem discussed earlier when I tried to add to the list.

What Is Known About the Unknown Type?

The wildcard is a good step forward, but it does not solve all my problems. You can see this if you think about what I can do with my list elements now. What if my `Person` superclass had a method `printAddressDetails` that I want to use as part of my `printList` method:

```
private void printList(List<?> list) {
    for (Person p: list) {
        ...
        p.printAddressDetails();
    }
}
```

This will now not work. The advantage of using the unknown type is that you can pass in lists of any type, but you pay by virtue of the fact that you don't know much about that type. All you know, in fact, is that it is a subtype of `Object` (because every type is a subtype of `Object`). So I cannot treat element types as `Person`s.

Not knowing much about the element type can still be OK in some cases. I could still use all list operations that do not depend on the element type, such as `size()` and `clear()`. I could also do anything that I can do with the `Object` type, such as using the `toString` method (maybe implicitly by calling `System.out.println`).

But to call type-specific methods, I need something else. In using the wildcard, I went from saying that my parameter is exactly a *List of Person* to saying that it is a *List of anything*. Instead, I would like to say that it is a *List of any subtype of Person*. I can do this with a *bounded wildcard*.

Bounded Wildcards

Generic parameters can have bounds, which restrict what kind of actual types can be used for them. Consider this next version of my `printList` method:

```
private void printList(List<? extends Person> list)
```

This definition now allows lists of `Person` or subtypes of `Person` (and only these) as parameters, just as I intended. Because I am using a wildcard, I am still not allowed to add to the list, but I know that all elements are of type `Person` (or its subtypes). I can now treat elements as `Person` objects and call the appropriate methods. This finally solves my problem.

Other Bounded Types

Wildcards are not the only place where bounds can be used and are useful. Type bounds can also be employed in the declaration of generic types and in methods without wildcards. For example, I can define a generic type `PersonList` that accepts only `Person` and its subtypes as parameters:

```
class PersonList<T extends Person>
```

This is similar to the definition of `ArrayList` that I showed in the last issue of *Java Magazine*, but this time only subtypes of `Person` can be used to instantiate the type:

```
PersonList<Student> students =
    new PersonList<Students>();
PersonList<Faculty> professors =
    new PersonList<Faculty>();
```

In return, all methods from the `Person` type can now be used on objects of type `T` in my implementation of the `PersonList` class, because I have a guarantee that any concrete instantiation of `T` will have these methods.

Generic Methods

This is a good time to introduce another generic feature: generic methods. In the previous examples, the generic type parameter was introduced in the class header when we declared a generic class. It is also possible to have single generic methods, without making the whole class generic. In that case, the single method can handle generic types. Generic methods are often combined with bounded generic types.

Consider the following example. Here, I attempt to write a method that prints all elements from a list that are smaller than a given limit:

```
public <T> void underLimit(List<T> myList, T limit) {
    for (T e : myList) {
        if (e < limit)
            System.out.println(e);
    }
}
```

The new syntax here is the type parameter `<T>` in the header after the keyword `public` and before the return type. I am assuming that this method is in a class that is not generic, so no type parameter has previously been declared. To use a generic type in the parameter list, I need to declare this type first—that is the effect of writing the type `<T>` in the header.

This code will fail, however, because the less-than operator cannot be applied to any unspecified type `T`. Instead, I can use the `compareTo` method, but this works only when `T` is a subtype of `Comparable`. I can enforce this by changing my method as follows:

```
public <T extends Comparable<T>> void underLimit(
    List<T> myList, T limit) {
    for (T e : myList) {
        if (e.compareTo(limit) < 0)
            System.out.println(e);
    }
}
```

Here, I have declared that I only accept types for type `T` that are subtypes of `Comparable` so that the methods needed are guaranteed to be available.

Upper Bounds and Lower Bounds

So far, I have discussed bounded types only by showing an upper bound to establish a supertype (an upper bound) for the wildcard parameter, for example:

```
List<? extends Person>
```

The effect is that only the named type or its subtypes can be used to instantiate the type. In other words, the concrete type at the point of use must extend (or implement) `Person`. If we were to draw a typical inheritance hierarchy around `Person`, only `Person` or the classes below it in the hierarchy can be used.

I can also restrict the type in the other direction, by declaring a lower bound:

```
List<? super Person>
```

By using the `super` keyword for my declaration, I am stating that the type has to be `Person` or a supertype of `Person`. If you picture this in an inheritance hierarchy, you can use `Person` or the types above it in the hierarchy. This is less often used than upper bounds but can be helpful in some situations.

Implementation

In addition to knowing how to use generic types, it is also useful to know a little bit about how they are implemented in the Java compiler and the JVM. If you ever talked with anyone about the implementation, it is likely that the term *type erasure* came up at some stage. It is important to know what this means, because it affects not only the efficiency of implementation but also the semantics of your code in certain cases.

Type Erasure

At the core of type erasure is the fact that type parameters exist only at compile time; they are completely removed at runtime. They are a construct exclusively used for type checking during compilation to ensure type safety, but they are not carried through into the Java bytecode.

To understand generics at first, it is often helpful to think of generic classes as expanded at instantiation time. For example, consider the following type:

```
class List<T> {
    public void add(T elem);
    ...
}
```

If it is then instantiated by using the concrete type `List<String>`, it can be thought of as having every occurrence of `T` in the source text replaced by `String`, so that the parameter type in the `add` method becomes `String`. For `List<Integer>`, each `T` would be replaced by `Integer`, and so on.

This is a useful mental model to start understanding generics, but it is ultimately false. It is useful, because it is easy to understand, and it gives a good approximation of how generics behave. It is important to know, however, how things really work, because sometimes that makes a noticeable difference.

Generic types are never expanded into their concrete instantiations: not in source code, not in binary code, not on disk, and not in memory. This is different than templates in C++, for example, where this expansion actually happens. In Java, the generated code will just insert `Object` as the type for each unbounded type parameter, or the bounding type for types that have bounds. Thus, `List<String>`, `List<Integer>`, and `List<Person>` are all represented by a single class `List<Object>` by the time your program executes. By then, the compiler has made sure that you used the class in a type-safe manner, and type problems have been prevented. You used many types but get only one class.

Discarding type parameter information at runtime has advantages and disadvantages. One of the advantages is that it saves time and space: the class file needs to exist only once for every generic class. It does not need to be stored or compiled multiple times. This is a clear benefit.

On the downside, type erasure makes life harder for tool writers, such as creators of development environments. It is hard, for example, for a debugger to figure out the correct type for an object at runtime if that type is derived from a generic class. No information is kept in the class file about the full type information.

More important for you as a programmer is the fact that type erasure can influence the behavior of your code. The following sections describe examples where it is necessary to understand type erasure to understand the behavior of the Java system.

No instanceof for Types with Type Parameters

The `instanceof` operator cannot be used with parameterized types. Consider the following attempt to use `List<T>` as defined in the previous section:

```
if (list instanceof List<Person>) {
    List<Person> pl = (List<Person>) list;
}
```

This code looks entirely reasonable, but if you consult the previous section on type erasure, you will see why it does not work: the runtime system has no idea whether a type is `List<Person>`, because it does not keep this information around. (All it knows about is `List<Object>` but nothing more specific.) So it cannot perform this check and give you the answer. You will see an error saying `illegal generic type for instanceof`.

The same problem shows up when you use the `getClass` method:

```
List<Student> sl = new ArrayList<Student>();
List<Faculty> fl = new ArrayList<Faculty>();
if (sl.getClass() == fl.getClass())
    ...
```

At first glance, you might think that the condition in the if-statement is false, but because of type erasure, it will actually evaluate to true. As far as the runtime system is concerned, the class of both objects is `ArrayList`.

Generic Classes and Static Attributes

One of the areas where type erasure becomes most visible in source code is when you use static attributes in generic classes. Static methods and static fields are shared between all instantiations of a generic class. The reason is again the same: only one copy of the generic class actually exists. You have to be aware of this to write correct code. A side effect of this is that it is not possible to declare a static field of a generic parameter type:

```
class MyClass<T> {
    private static T value;   // error
    ...
}
```

Because this field is shared between all variants of the type, it cannot refer to the type parameter of specific instantiations.

Java Trivia: Arrays and Type Safety

If you are interested in the details of Java and type safety, you might like this little bit of Java trivia: the implementation of arrays in Java has a hole in its type system. This is one of the rare cases where Java is not statically type-safe.

The problem is the same problem I discussed earlier in this article: If `B` is a subtype of `A`, is then `List<B>` a subtype of `List<A>`? For lists, the answer is no. Earlier in this article, I explained why this is and how it could go wrong if we were to consider `List<B>` a subtype. However, for arrays (a very similar situation), Java does consider the list to be a subtype. This introduces a potential type problem. Consider the following code:

```
A[] aa;
B[] ba = new B[3];

aa = ba; // allowed! B[] is subtype of A[]
aa[0] = new B();
aa[1] = new A();  // java.lang.ArrayStoreException: A
```

The last line in this example represents a type error: I am trying to insert an `A` object into an array of `B`. The problem is that the assignment in the third line is allowed. This problem

is picked up only at runtime, not at compile time, breaking Java's static type safety. When it designed generic classes, the Java team decided to be more conservative and detect the equivalent problem at compile time.

Conclusion

Generic types are easy to understand in principle and generally quite easy to use. However, when you start writing more sophisticated code—particularly if you're writing libraries—you might run into a whole range of situations where you need to understand the advanced constructs in generics.

When you put all of the concepts together, the class and method definitions can become quite tricky to read even for experienced programmers. Have a look at the `max` method of class `Collections` in the standard library, for example, or the definition of methods in the `Class` class. You will see that it can take some time to get your head around the combination of all the constructs. Do not let this discourage you; these complex constructs are rare, and with the concepts I have discussed here and some practice, you should be able to work out most of it. More importantly, you should be able to write correct and flexible code yourself. `</article>`

Michael Kölling is a Java Champion and a professor at the University of Kent, England. He has published two Java textbooks and numerous papers on object orientation and computing education topics, and he is the lead developer of BlueJ and Greenfoot, two educational programming environments. Kölling is also a Distinguished Educator of the ACM.

learn more

The Java Tutorial on generic types

FEATURED JDK ENHANCEMENT PROPOSAL

JEP 282 jlink: The Java Linker

For most readers, the idea of a linker for Java might seem very peculiar indeed. Linker functions, which are part of a build tool associated with native languages, are performed by the JVM in its class-loading mechanism. In particular, these functions are executed in the algorithms for finding JARs that contain referred-to classes and methods and then loading them into the current JVM memory space. [For more information on this process, download a PDF of our article "How the JVM Locates, Loads, and Runs Libraries" by Oleg Šelajev. —*Ed.*]

What JEP 282 proposes is not the traditional linker but, rather, a generic tool that runs where a linker does in the build process—after the compiler but before creation of the executable. The tool would define a plugin interface, by which a variety of tools could be inserted into the build process. The most obvious of these would be an optimizer, especially a whole-program optimizer that could identify opportunities to improve performance and reduce code size that are not visible to the compiler on a class basis. Other plugins suggested in the JEP document could remove debug information, reorder resources so that they can be loaded faster, and even compress generated files.

In theory, many other refinements to generated code could be performed—including those from third parties. Some examples are insertion of instrumentation data, supplementation of debugging data, conversion of bytecodes to other formats, intraclass optimization, and so on. All of this could be done through plugins to the proposed jlink technology.

CHARLES NUTTER

JRuby 9000: Beautiful Language, Powerful Runtime

A simple language that inspired Ruby on Rails and can greatly facilitate complex Java coding, such as JavaFX development, using native libraries

In May, the JRuby team released JRuby 9.1, the latest version in the JRuby 9000 line. The team put a lot of hard work into making JRuby 9000 the best implementation of Ruby available. As a member of the team, I will demonstrate why you might want to take a look at Ruby on the JVM, specifically using JRuby.

What Is Ruby?

Ruby is a dynamically typed, object-oriented language inspired by Smalltalk, Perl, and Lisp. It was created in 1995 by Yukihiro "Matz" Matsumoto; Ruby 2.3 is the current version. Over the past 10 years, it has become one of the top 10 languages in use, driven in part by the success of the Ruby on Rails web framework. These days, Ruby is used by some of the biggest companies in the world, and not just for web development.

Unfortunately, the standard implementation of Ruby—usually referred to as CRuby or MRI (Matz's Ruby Implementation)—lacks some features modern developers want and often need such as a high-speed just-in-time (JIT) compiler; scalable, low-pause garbage collection; and true parallel execution. That's where JRuby comes in.

What Is JRuby?

JRuby is an implementation of Ruby atop the JVM, written mostly in Java (but a growing amount in Ruby) and supporting 99 percent of Ruby features. As much as possible, the JRuby team has tried to ensure that JRuby remains compatible with CRuby, all while leveraging the JVM's power.

JRuby's garbage collector is the JVM's garbage collector, and there are a lot of excellent garbage collectors available for today's JVMs. [For a comparison of several JVM garbage collectors, see "The New Garbage Collectors in OpenJDK" by Christine Flood in the March/April issue of this year. —*Ed.*] JRuby's threads are JVM threads, which means true parallel execution of Ruby code. JRuby compiles Ruby code to JVM bytecode, which the JIT can then compile to native machine code. In fact, JRuby was the first Ruby implementation to have any native JIT capabilities.

These features all combine to create an extremely powerful tool: all the beauty and fun of programming Ruby with the best of the JVM. So, what can you do with this tool?

Getting Started

JRuby, like most JVM-based libraries and applications, is distributed in a number of prebuilt binary forms.

Most users will want a full JRuby distribution, available at http://jruby.org, which includes command-line utilities (like the `ruby` and `gem` commands), the Ruby standard library, and a filesystem layout similar to CRuby. I recommend the use of so-called "Ruby switchers" such as RVM, which will down-

load and install the latest JRuby implementation.

```
$ rvm install jruby-9.1.2.0
Searching for binary rubies, this might take some time
Found remote file /Users/headius/.rvm/...
Checking requirements for osx.
Requirements installation successful.
jruby-9.1.2.0 - #configure
jruby-9.1.2.0 - #download
jruby-9.1.2.0 - #validate archive
jruby-9.1.2.0 - #extract
jruby-9.1.2.0 - #validate binary
jruby-9.1.2.0 - #setup
...
```

See RVM's home page for more details, or try out one of the other Ruby switchers.

For folks who prefer a more direct approach, you can simply download a tarball (.tar.gz) or a zip file containing a full JRuby distribution. Unpack it, add the `bin` directory to your `PATH`, and you're off to the races. JRuby also comes in a Windows installer that can optionally install a JRE for you as well.

JRuby also publishes a full complement of Maven artifacts under the "org.jruby" group, which is useful for embedded applications that will not need a complete on-filesystem JRuby distribution.

Once installed, JRuby's command line matches CRuby's:

```
$ jruby -v
jruby 9.1.2.0 (2.3.0) 2016-05-26...
$ jruby -e "puts 'Hello, Ruby'"
Hello, Ruby
```

And Ruby's interactive console, IRB, is available as well:

```
$ irb
jruby-9.1.2.0 :001 > puts "hello"
hello
...
```

Ruby on Rails

Every web developer should have heard of Ruby on Rails by now. It changed the way developers do web development, from introducing the idea of sensible defaults (convention over configuration) to rich code generation (scaffolding) and database-agnostic schema versioning (migrations). Most web frameworks today copy some aspect of Rails, in some cases even mimicking the filesystem layout of Rails applications or reusing Rails-inspired terms for similar features.

Next, I walk through getting a simple Rails app running on JRuby.

But first, I need to install Rails using the `gem install` command. Most libraries for Ruby are distributed as "gems" hosted on RubyGems.org. From a Java perspective, think of a Ruby gem as a Maven library plus some executable scripts for the command line.

```
$ gem install rails
Fetching: rack-1.6.4.gem (100%)
Successfully installed rack-1.6.4
Fetching: sprockets-3.6.0.gem (100%)
Successfully installed sprockets-3.6.0
...
Fetching: rails-4.2.6.gem (100%)
Successfully installed rails-4.2.6
23 gems installed

$ rails new my_app
      create
      create  README.rdoc
      create  Rakefile
...
```

```
      create  vendor/assets/stylesheets
      create  vendor/assets/stylesheets/.keep
         run  bundle install
Fetching gem metadata from https://rubygems.org/
Fetching version metadata from https://rubygems.org/
Fetching dependency metadata from https://rubygems.org/
Resolving dependencies.............
Using i18n 0.7.0
Using json 1.8.3
Installing minitest 5.9.0
...
Installing sass-rails 5.0.4
Installing turbolinks 2.5.3
Bundle complete! 11 Gemfile dependencies,
                 54 gems now installed.
Use 'bundle show [gemname]' to see where
             a bundled gem is installed.
```

[Due to width constraints, some lines in this output and in other output shown in this article have been truncated, folded, or had unnecessary data removed. —*Ed.*]

Now the magic of Rails starts to kick in. By using the `rails new` command, I get a fully functional, bare-bones application, complete with a welcome page, a convention-based filesystem layout, and a basic database configuration using sqlite3 (you can specify a different database with the `-d` flag). Rails constructs the application and then runs the `bundle` command. Bundler is a gem-based dependency management tool for applications; Rails builds a Gemfile containing a list of all libraries required for the app, and Bundler makes sure they're installed.

At this point, I can start up the Rails application, even though I haven't written any logic.

```
$ cd my_app
$ rails server
=> Booting WEBrick
=> Rails 4.2.6 application starting in
     development on http://localhost:3000
=> Run 'rails server -h' for more startup options
=> Ctrl-C to shutdown server
[2016-06...] INFO  WEBrick 1.3.1
[2016-06...] INFO  ruby 2.3.0 (2016-06-06) [java]
[2016-06...] INFO  WEBrick::HTTPServer#start:
                    pid=37393 port=3000
```

Let's quickly scaffold some basic functionality for our application. In Rails, *scaffolding* is code generated at development time that provides a rough structure for your application. You can tell Rails to generate models, views, controllers, tests, and more. The following example generates the basic code for CRUD operations against a "post" with a "title" and a "body."

```
$ rails generate scaffold post title body:text
   invoke  active_record
   create
       db/migrate/20160606083900_create_posts.rb
   create    app/models/post.rb
   invoke    test_unit
   create      test/models/post_test.rb
   create      test/fixtures/posts.yml
   invoke  resource_route
   route    resources :posts
   invoke  scaffold_controller
   create    app/controllers/posts_controller.rb
   invoke    erb
   create      app/views/posts
   create      app/views/posts/index.html.erb
   create      app/views/posts/edit.html.erb
   create      app/views/posts/show.html.erb
   create      app/views/posts/new.html.erb
   create      app/views/posts/_form.html.erb
```

```
   invoke    test_unit
   create
       test/controllers/posts_controller_test.rb
...
```

In addition to the actual application code and tests, the scaffolding generated what's called a *database migration*. This is a short script that can take one version of the database scheme and apply changes needed to migrate to the next version. These migrations allow you to roll schemas back and forth in a database-agnostic way.

Then I just need to roll the database migration forward and start the server again.

```
$ rake db:migrate
== 20160606083900 CreatePosts: migrating =========
-- create_table(:posts)
   -> 0.0075s
   -> 0 rows
== 20160606083900 CreatePosts: migrated (0.0099s)

$ rails server
=> Booting WEBrick
...
```

Here I am using the rake command, which is roughly equivalent to using Ant or Maven, minus the dependency management. Once Rails has migrated to the latest database schema, I can start up the server and, presto, I have a basic web GUI for CRUD operations.

Rails is still the killer app for Ruby, and if you haven't tried it before, perhaps JRuby can be your excuse to try it now. Check out the excellent documentation and tutorials on the Rails site, or pick up one of the many Rails books out there.

OK, I've built a killer app. Now, how do I deploy it with JRuby?

JRuby supports two-way integration with other JVM languages, so all those Java libraries you're familiar with can still be in your toolbox.

Deploying Rails on JRuby

In CRuby, if you want to handle any requests in parallel, you need to spin up separate processes— that is, completely independent VMs that share no resources. As a result, even small applications will consume more memory, and if they need to do any communication, you're forced to use some interprocess communication. Data sharing has to be done in a third process, such as a database or memcached, because those processes share only read-only application structure. Now you have a whole bunch of Ruby virtual machines running, each with its own heap and garbage collector—this is not the best use of resources in this multicore era.

In JRuby, you can take that same Rails application and handle your entire load inside a single process, with a single garbage collector tuned for concurrency and scalable heaps. That one process can be a standalone server, or you can deploy "JRuby on Rails" as a Java WAR file to any standard web container such as Tomcat or WildFly. Whether you're coming to JRuby from Ruby or Java, deployments of JRuby applications fit your world and make better use of your hardware.

For simple, standalone use, the Puma gem, which is the most popular pure-Ruby web server, is generally recommended.

```
$ gem install puma
Fetching: puma-3.4.0-java.gem (100%)
Successfully installed puma-3.4.0-java
1 gem installed
$ puma
Puma starting in single mode...
* Version 3.4.0 (jruby 9.1.3.0-SNAPSHOT - ruby 2.3.0),
```

```
     codename: Owl Bowl Brawl
* Min threads: 0, max threads: 16
* Environment: development
* Listening on tcp://0.0.0.0:9292
Use Ctrl-C to stop
```

If you need to deploy to an existing Java app server or web container, use the Warbler gem to package your Rails app (plus all its dependencies) into a deployable WAR file.

```
$ gem install warbler
Fetching: rubyzip-1.2.0.gem (100%)
Successfully installed rubyzip-1.2.0
Fetching: jruby-rack-1.1.20.gem (100%)
Successfully installed jruby-rack-1.1.20
Fetching: jruby-jars-9.1.2.0.gem (100%)
Successfully installed jruby-jars-9.1.2.0
Fetching: warbler-2.0.3.gem (100%)
Successfully installed warbler-2.0.3
4 gems installed
$ warble
rm -f my_app.war
Creating my_app.war
```

That's all there is to it.

Scripting Java

There's another feature of JRuby that makes it even more attractive for Ruby and Rails developers: you can call any library on the JVM as if it were just another piece of Ruby code.

JRuby supports two-way integration with other JVM languages, so all those Java libraries you're familiar with can still be in your toolbox. In fact, scripting Java libraries with JRuby is often much more fun and much easier than writing Java code. Let's have a look at a few examples.

```
java_import java.lang.System

Frame = javax.swing.JFrame
Button = javax.swing.JButton
Label = javax.swing.JLabel

frame = Frame.new("Java Home Checker")
button = Button.new("Display Java Home")
label = Label.new

button.add_action_listener do
  label.text = System.get_property('java.home')
end

frame.content_pane.layout = java.awt.FlowLayout.new
frame.content_pane.add(button)
frame.content_pane.add(label)

frame.set_default_close_operation(Frame::EXIT_ON_CLOSE)
frame.set_size(500, 100)
frame.visible = true

sleep
```

This simple example already shows some of JRuby's advantages. Specifically, imports are just plain Ruby code. The code shows two ways to import a class: using the `java_import` function or simply using the fully qualified long class name (and assigning it to a short one).

Java method names are tweaked a bit to make them look more like Ruby method names: snake_case is used instead of camelCase, set/get properties can omit set/get and be called by just the attribute name, parentheses are optional, and so on. In fact, you might not even know this code calls a Java library if you weren't familiar with Swing.

Simple interfaces can be implemented on the fly by passing

a block of code, similar to Java 8 lambdas. But JRuby can also dynamically add an interface implementation to any object, and it doesn't have to implement all methods (usually by including a `method_missing` fallback).

There's a lot less noise and ceremony in this code than there would be in the Java version.

Let's take a look at a few more-advanced examples of what you can do with JRuby's Java integration.

JRubyFX

In late 2008, Sun Microsystems released the first version of JavaFX, a new GUI toolkit inspired by web technologies and destined to be the replacement for Swing. JavaFX initially had its own language, JavaFX Script, but in an already tight world of language options, only the GUI Toolkit survives to this day. That means you'll be writing your JavaFX logic in Java. Perhaps a little JRuby can help here, too.

Enter JRubyFX, a Ruby API and wrapper for writing JavaFX applications. Let's walk through a simple example.

```
01 require 'jrubyfx'
02
03 class HelloWorldApp < JRubyFX::Application
04    def start(stage)
05        stage.title = "Hello World!"
06        stage.width = 800
07        stage.height = 600
08        stage.show()
09    end
10 end
11
12 HelloWorldApp.launch
```

On line 1, I require `jrubyfx`, which is a set of bindings for the JavaFX library, so I can use its features. In Ruby, libraries are brought into the process using `require`. Generally these libraries are installed on the local filesystem as Ruby sources, but occasionally they will bring along extensions written in other languages.

On line 3, I have a Ruby class definition extending the `JRubyFX::Application` class. Classes in Ruby are declared with the `class` keyword, just as in Java, but instead of having an `extends` keyword Ruby uses the less-than symbol (<). The double colons are Ruby's way to indicate namespacing.

The first method definition is on line 4. Because Ruby is dynamically typed, there are no type declarations for a method return or method parameters.

Lines 5 through 8 set up the stage. I set a title and window size (using Ruby-style attribute assignment rather than Java's "set" methods), and then I tell JavaFX to show the stage.

Lines 9 and 10 end the method and the class definition. Most lexical scopes in Ruby are closed with the `end` keyword, although short blocks (lambdas) frequently use curly braces.

And finally, on line 12, I tell the new JRubyFX application to launch itself. It's that easy.

Now let's look at how Ruby can really make the application fun and easy to write.

```
def start(stage)
    with(stage, title: "Hello World!") do
        layout_scene(800, 600) do
            label("Hello World!")
        end
    end
    stage.show # you can also put the
               # method call inside the block
end
```

Here, the `start` method is a bit more complicated. The most obvious change is the call to `with`, which takes a block of code using JRubyFX's scene-building DSL. Given the `Stage` provided by JavaFX and a title, I proceed to build the stage's con-

tents. I tell the DSL how to lay out this scene, and then I add some actual content: a JavaFX label.

What about FXML, JavaFX's XML-based markup for describing scenes? Building the scene with Ruby is great (and certainly a lot less hassle than doing it in Java), but for larger applications, you probably want a description of the GUI that's separate from the application code.

Here's an FXML definition for my simple scene:

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.util.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.paint.*?>
<?import javafx.scene.text.*?>

<HBox alignment="CENTER"
      xmlns:fx="http://javafx.com/fxml">
  <children>
    <Label text="Hello World!!" underline="true">
      <font>
        <Font size="66.0" />
      </font>
    </Label>
  </children>
</HBox>
```

Using this definition in JRubyFX is as simple as adding a call to the `fxml` method, as shown next.

```
def start(stage)
    with(stage, title: "Hello World!",
                width: 800, height: 600) do
      fxml "Hello.fxml"
      show
    end
end
```

JRuby and JavaFX work really well together, so if you haven't had a chance to try JavaFX, JRubyFX might be the most fun you'll have this week. Check out the complete Getting Started page for JRubyFX.

Beyond the JVM

JRuby strives to be pure Java (and some Ruby) as much as possible, and it supports users on a wide array of platforms, from Linux to OpenVMS. The platform-independence of Java serves you well here. Unfortunately, that independence sometimes means you can't integrate with the host platform as well as a native application can (or, in this case, as well as CRuby can).

To maintain JRuby's high level of compatibility, you often need to call out to native libraries. Normally on the JVM this would mean writing a lot of Java Native Interface (JNI) code for every function to be called from Java, building the code for all supported platforms, and shipping that ever-growing binary with JRuby. That approach obviously doesn't scale, so the JRuby team took a different approach: it uses the Java Native Runtime (JNR) to load and bind libraries dynamically at runtime.

JNR—similar to Java Native Access (JNA), which you might already be familiar with—uses a low-level binding for libffi (the foreign function interface [FFI] library used by most UNIX platforms) to pull a library in, find the needed function, and bind it to a Java interface. JRuby pulls in and binds a large number of POSIX functions, UNIX socket support, native I/O file descriptors, and much more.

As a Rubyist, you can also leverage JRuby's native support via the ffi gem, which provides an easy-to-use Ruby API to call native libraries.

```
require 'ffi'

module POSIX
  extend FFI::Library
  attach_function :getuid, :getuid, [], :uint
  attach_function :getpid, :getpid, [], :uint
end

pid, uid = POSIX.getpid, POSIX.getuid
puts "Process #{pid} running as user #{uid}"
```

In this example, I've created a Ruby module to hold native function bindings. Think of a module as an interface with default implementations for every method. Those methods can be class methods (similar to static methods in Java) or instance methods that are added to a class hierarchy when the module is included (similar to implementing an interface in Java).

Inside the `LibC` module, in the next listing, I extend the `FFI::Library` module, which injects other FFI methods I can use to bind functions and define native data types. Now I have access to `attach_function` from the previous listing, which takes as arguments the name of the function I want to call, an optional Ruby name to assign to the function, and information about parameter types.

That's it. Run this code on JRuby, and you'll see the real, live process ID and user ID for the host JVM—something that's not possible to do with pure Java code.

```
class Timeval < FFI::Struct
  layout :tv_sec => :ulong, :tv_usec => :ulong
end

module LibC
  extend FFI::Library
  attach_function :gettimeofday,
      [ :pointer, :pointer ], :int
end

t = Timeval.new
LibC.gettimeofday(t.pointer, nil)
puts "t.tv_sec=#{t[:tv_sec]} \
      t.tv_usec=#{t[:tv_usec]}"
```

JRuby's FFI also provides a way to define native data types, such as structs. In the preceding code, I define a `Timeval` struct that has an in-memory layout of two unsigned longs: `tv_sec` and `tv_usec`. I bind in the libc `gettimeofday` function, construct a new instance of `Timeval`, and make the call. The native call populates a native struct that I can then read from like a normal Ruby object, all without writing a line of C code. Pretty cool, right?

FFI is capable of much more than this, and there are many large production apps out there leveraging JRuby's native capabilities. For more information, stop by the Ruby FFI project.

The Future of JRuby 9000

JRuby 9000 represents one of the most advanced JVM language implementations available. It has its own bytecode-like intermediate representation, an optimizing compiler, and a mixed-mode interpreter plus a JIT compiler (very much like the JVM itself). The JRuby team has been pushing the limits of what a language can do atop the JVM. In fact, JRuby is currently the fastest Ruby implementation available. By the end of this year, the team hopes to utilize its internal representation (IR) runtime to make all JRuby code perform comparably to equivalent Java code, without sacrificing any of Ruby's unique features.

But the JRuby team is not stopping there. In late 2014, the team partnered with Oracle Labs to open-source their Truffle-based Ruby implementation as part of the JRuby project. Truffle is a next-generation language runtime that

lives alongside JVM bytecode but it uses the pure-Java Graal JIT compiler to directly optimize a language's behavior. As a result, JRuby plus Truffle might prove to be the fastest way to run Ruby on any runtime, albeit with the requirement that you run on a Graal-friendly JVM such as JDK 9. The JRuby team hopes to see the JRuby plus Truffle runtime production-ready in the next couple of years.

The team is very excited, both about its IR runtime and about Truffle's potential. Ruby is no longer a slow language.

Conclusion

Ruby is a beautiful, fun language with a rich ecosystem and a friendly, helpful community. That community has built Rails into the powerhouse it is today—the fastest way to get a well-structured web application deployed to production. You can leverage the best of the Ruby world and the best of the Java world using JRuby—deploying to the same servers, using the same libraries, getting the best out of the JVM—and you just might have fun doing it. It's a great time to try JRuby. `</article>`

Charles Nutter is a Java Champion who works at Red Hat on JVM languages and bending the JVM to his whims. He has been a co-lead of the JRuby project for the past 10 years, and worked as a lead Java EE architect for many years before that. He hopes to keep the Java platform open and evolving, and works to expand the platform to new languages and new ways of building software.

learn more

- Home of the Ruby language (non-JVM)
- Truffle on the JDK

BUCHAREST JUG

Bucharest, Romania, is a regional leader in software development. The Bucharest Java User Group was formed to create a strong community for all the developers in Bucharest who are using Java- and JVM-based programming languages.

The first meeting took place in May 2012 with approximately 25 participants. The Java user group (JUG) is now led by Alex Proca and Alin Pandichi and has more than 600 registered members. It organizes monthly meetings with one or two presentations, starting around 7 p.m.; later on, it moves to a pub for drinks. Occasionally, it hosts hands-on labs such as the recent workshops on the MVC 1.0 (JSR 371) Java EE specification and on JavaFX. Around 50 participants usually attend the talks, and 10 attend the workshops.

The speakers are often selected from the local pool of talented Java developers—for instance, Eugen Paraschiv (also known as Baeldung), whose tutorials and reviews have garnered a sizable following. From time to time, the JUG hosts international speakers such as Java Champion Axel Fontaine, who gave a presentation about immutable infrastructure.

Local interest in Java, catalyzed by the JUG, led to a Java conference, Voxxed Days Bucharest, which was first held in March 2016. The organizers are already looking forward to next year's event.

The Bucharest JUG keeps in close contact with members of the worldwide Java communities. Contact it via email or follow it on Twitter, Facebook, Google+, or Meetup.

SIMON ROBERTS

Quiz Yourself

More subtle questions from an author of the Java certification tests

I've put together more interesting problems that simulate questions from the 1Z0-809 Programmer II exam, which is the certification test for developers who have been certified at a basic level of Java programming knowledge and now are looking to demonstrate more-advanced expertise. [Readers wishing basic instruction should consult the "New to Java" column, which appears in every issue. —*Ed.*]

Question 1. Given this class declaration:

```
public class Tire { private int diameter, width; }
```

Which two actions are normally performed to support simple use in the Collections framework? Choose two.

a. Add a method with the signature `public boolean equals(Tire t)`.
b. Add a method with the signature `public int hashCode(Tire t)`.
c. Add a method with the signature `public boolean equals(Object o)`.
d. Add a method with the signature `public int hashCode()`.
e. Arrange that the class implements `Comparable<Tire>`.

Question 2. Given the following code:

```
public class BaseException extends Exception {}
public class OneException extends BaseException {}
public class TwoException extends BaseException {}
public class ThreeException extends Exception {}

public class MultiCatch {
  public void fingersCrossed()
    throws OneException, TwoException,
                ThreeException { }

  public void tryThingsOut() /* Point A */{
    try {
      fingersCrossed();
    } catch (OneException | TwoException ex) {
      ex.printStackTrace();
      throw ex;
    } catch (ThreeException e) {
      e.printStackTrace();
    }
  }
}
```

Which is the best change? Choose one.

a. No change is necessary; the code is ideal as shown.
b. The code should be modified by adding at `/* Point A */` the text `throws Exception`.
c. The code should be modified by adding at `/* Point A */` the text `throws BaseException`.
d. The code should be modified by adding at `/* Point A */` the text `throws OneException, TwoException`.
e. The code should be modified by adding at `/* Point A */` the text `throws OneException, TwoException, ThreeException`.

Question 3. Given the following code:

```
System.out.println(
  Stream.empty().findAny()
  // Line n1
);
```

Which two, applied independently, may be added at line n1 to cause the output `"Empty"`**?** Choose two.

a. `.ifPresent(s->s).orElse("Empty")`
b. `.orElse("Empty")`
c. `.orElseGet(() -> "Empty")`
d. `.orElseGet("Empty")`
e. `.orElseSupply(()->"Empty")`
f. `.otherwise("Empty")`

Question 4. Which of the following two statements, independently, might be good uses of assertions? Choose two.

a. `assert x >= 0 : "X must be non-negative";`
b. `assert x++ > 0;`
c. `assert x == 0;`
d. `assert (++x > 0 , "X must be non-negative");`
e. `assertTrue "X must be zero" : x == 0;`

Question 1. The correct answers are options C and D. In this question, there are three methods to choose among: `equals`, `hashCode`, and the `compareTo` method of the `Comparable` interface. While order comparisons are certainly relevant to some parts of the Collections API—notably those that actually depend on ordering, such as `TreeSet`—order isn't really a fundamental part of the API as a whole. However, the idea of equality is absolutely fundamental. The basic way that most collections determine whether they contain one particular object is by using the `equals` method to see whether the object in the collection is equivalent to the one being asked about.

So, `equals` is almost certainly the first method you'll think about implementing for any class that's going to be stored in a collection, at least if there's any chance of needing to find it directly. The next question is which of the two proposed method signatures is correct. Both will compile, but the actual signature must take an `Object` argument. There are two reasons. First, from a syntax perspective, this method must override the `equals` method defined in `java.lang.Object`, and that's defined to take an `Object` argument. Second, from a philosophical perspective, it's perfectly reasonable to ask whether "this apple is equal to that banana." The answer is simply "no." It's tempting in these days of familiar generics to think that the method would take an argument of the object's own type, but if that method is implemented, it will be ignored by the Collections API. So, option A is incorrect, and option C is the proper `equals` method. Don't forget that when overriding a method, it's good practice to use the `@Override` annotation. That will ensure that if you declare the argument as anything other than `Object`, the compiler will tell you that you made a mistake.

Next, the documentation for `equals` states, "Note that it is generally necessary to override the `hashCode` method whenever this method is overridden, so as to maintain the general contract for the `hashCode` method, which states that equal objects must have equal hash codes."

Given this requirement, it's pretty clear that implementing `equals` almost mandates implementing `hashCode`. The remaining question is what the signature should be. Given

that this is an instance method, and it generates a representative number based on the contents of this, it's fairly clear that no argument is needed, and indeed, the documentation shows that option D is correct and option B is wrong.

It's fair to note that although `equals` and `hashCode` are the most fundamental methods required of classes that participate in collections, the `Comparable` interface, with its method `compareTo`, is certainly relevant in some situations. Ordered structures, such as the implementations of `SortedSet` and `SortedMap`, often use it. If this were an exam question, the fact that you're told to select two answers should remove any small doubt you might have had about leaving option E unchecked. It's worth mentioning that it's a matter of policy: Oracle's Java certification exam questions always state exactly how many answers you should select. Be careful not to throw points away by ignoring this advice!

Question 2. The correct answer is option D. The essence of this question has two parts. First, the exception classes are all checked exceptions, which means that if you rethrow the exception caught in the multicatch (that is, in the part `catch (OneException | TwoException ex)`), you must declare that the method throws that exception type. This means that option A is incorrect; the code does not compile as is.

The second part of the question relates to what the type of the formal parameter `ex` actually is, and what the checked exception mechanism demands for the declaration. This is slightly trickier, and it's where that bothersome word *best* in the phrase "the best change" comes in.

In the code, `ex` can really have only one type, and this is actually the closest common parent of the types listed in the multicatch; that is `BaseException`. However, the checked exception mechanism understands the multicatch syntax, and when rethrowing `ex`, only those exceptions specifically listed in the `catch` parameter list need to be declared in the `throws` clause. Option D lists exactly the same exceptions as the `catch` block, and it is the correct answer.

Finally, there's the question of whether the other answers are "wrong" or whether they could be cause for a complaint to the examiners. Well, it's certainly true that options B, C, and E also compile. Functionally, they work perfectly well, too, and that might tempt you to believe that they're all equally valid. However, checked exceptions have two consequences.

First, they put a burden on the programmer who writes the calling code, and (except with interfaces) you should not declare more exceptions than you might throw without a good reason. Second, they should convey useful information about failure modes to the caller. If you declare a more general exception, that information is diluted or lost, which is unhelpful and reduces readability. Both of these reasons should tell you that options B, C, and E, all of which declare throwing more exception classes than are actually possible, are not as good as option D.

Of course, it's possible that you don't accept the reasoning just given. Java's checked exception mechanism is the subject of much debate, so it's clear that opinions differ. But most of those who dislike checked exceptions would strongly support the notion that throwing too many exceptions is bad. Anyway, the final observation is that you are told to choose one answer. So, you know that you must distinguish among four compilable answers. You need to find a plausible reason to make a choice, even if you don't personally like the reason.

Don't be afraid to apply a little logic to separate plausible answers from better answers. It's probably the case that this question would be subject to considerable scrutiny by the exam team. Indeed, I suspect it might be rejected. My purpose in including it here is to illustrate how logic and knowledge of good practices can be applied to choose one "right" answer among several answers that compile and run successfully.

Question 3. The correct answers are options B and C. This question hinges on some knowledge about `Stream` API behavior and the `Optional` class.

First, the `findAny` method returns an object from the

stream. However, the stream might be empty, and whenever there's a chance that a stream terminal operation might not have anything to return, the `Optional` class is used to represent the result. As a side note, `Optional` is an API mechanism intended to avoid null pointer exceptions. Tony Hoare, inventor of null pointers, has acknowledged that these have caused many bugs; he now refers to them as his "billion-dollar mistake." To be fair, the use of special values to indicate errors or exceptional situations is now almost universally recognized as a bad thing, and exceptions address this issue, too.

Once you recognize that you will get an `Optional` from the `findAny` terminal operation, you need to know how to interact with it and get the text "Empty" from our `println` method call. In this case, you know that the stream is empty and, therefore, `findAny` will return an empty `Optional` to us.

Given an empty `Optional`, there are two methods intended to directly return a value for your use. Consulting the API documentation, you can see that these methods are `orElse` and `orElseGet`. Both methods return the contents of the `Optional` if it is not empty or an alternative value if it is empty. The `orElse` version takes a simple value that is to be returned, matching the call in option B. The `orElseGet` version takes a `Supplier`, which is invoked in the event that the `Optional` is empty. `Supplier` is a functional interface that defines a method that takes no arguments and returns a value. To create that as a lambda expression requires the empty parentheses, followed by the arrow symbol, and then the expression that defines what the newly supplied value will be. That suggests the form `() -> expression`, or, to return the specific literal: `() -> "Empty"`, which is option C. The other options are syntactically incorrect.

Question 4. The correct answers are options A and C. Here's that word *good* again. It gives you a bit of a hint that some level of judgment beyond whether or not something compiles might be important here.

In this case, three options—options A, B, and C—will compile. Option D fails because the `assert` keyword is just that: a keyword. It's not a method call, so the syntax there is bogus. Also, `assertTrue` in option E is not part of the core Java SE API, but the name is used in tools such as JUnit. However, `assertTrue` in JUnit is a method, so it requires parentheses and a comma rather than a colon to separate its parameters. Option E is, therefore, wrong. As a side note, the Java exam is about core Java features, not third-party APIs, so if you knew what `assertTrue` is about, you should have rejected it from consideration for that reason.

Of the three compilable options, two are "good" and one is severely broken. Option A is the two-operand form of `assert`; the first operand is a boolean expression, and the second is a text message that will become the message in the `AssertionError` that is thrown if the boolean evaluates to false. Option C uses the single-operand form, which executes the boolean test and builds an `AssertionError` with a null message if the boolean evaluates to false. Next, let's look at why option B is a huge error, even though it compiles.

The goal of assertions is to allow the programmer to put certain statements about design intent into the code, in a way that forms documentation that cannot be wrong. The boolean expression that forms the required operand for an `assert` must be true; otherwise, the `assert` is expected to complain. These little statements can be very helpful when picking up code that someone else wrote; they can tell all sorts of useful details about how the code works. However, because the expression in the assertion seemingly must be evaluated every time the program runs past the statement, it's possible to be concerned that the CPU usage of all these little tests could adversely affect performance.

A performance concern like that would probably discourage most programmers from using assertions freely. However, assertions have a neat trick: the code of assert statements can be stripped from the bytecode during classloading. If this happens, the statements have zero performance impact.

It turns out that stripping them is the default behavior. So, you must explicitly use the command-line option `-ea`, or `-enableassertions`, for the assertions to be executed. (Unfortunately, IDEs also generally duplicate this default.)

The intention is that programmers always test their code with `-ea` in effect and that the user runs the final program without it. That creates an elegant "best of both worlds" situation that, in my view, gets far less use than it should.

Of course, this "conditional execution" also creates an interesting potential problem. Imagine that the boolean expression in your assertion actually does something of computational significance. It has a side effect and changes something in some way. Now you have the makings of a disaster; the functional behavior changes depending on whether you run in development mode (with `-ea`) or production mode (without it). The documentation of `assert` goes to great lengths to point out that side effects of any kind must be avoided in an assert statement. For this reason, option B is bad and, therefore, incorrect.

The *Java Language Specification*, in section 14.10, notes, "Because assertions may be disabled, programs must not assume that the expressions contained in assertions will be evaluated. Thus, these boolean expressions should generally be free of side effects." You can find more discussion on safe and appropriate uses of `assert` here. Notice that in that document, there are other do's and don'ts, which is why the question in this quiz asks "which might be good uses…," rather than "which are good uses…." Options B, D, and E cannot possibly be good; the other two might be if other conditions are met, but no information is available on those issues. `</article>`

Simon Roberts created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is now a freelance educator at many large companies. He remains involved with Oracle's Java certification.

//contact us /

Comments

We welcome your comments, corrections, opinions on topics we've covered, and any other thoughts you feel important to share with us or our readers. Unless you specifically tell us that your correspondence is private, we reserve the right to publish it in our Letters to the Editor section.

Article Proposals

We welcome article proposals on all topics regarding Java and other JVM languages, as well as the JVM itself. We also are interested in proposals for articles on Java utilities (either open source or those bundled with the JDK). Finally, algorithms, unusual but useful programming techniques, and most other topics that hard-core Java programmers would enjoy are of great interest to us, too. Please contact us with your ideas at javamag_us@oracle.com and we'll give you our thoughts on the topic and send you our nifty writer guidelines, which will give you more information on preparing an article.

Customer Service

If you're having trouble with your subscription, please contact the folks at java@halldata.com (phone +1.847.763.9635), who will do whatever they can to help.

Where?

Comments and article proposals should be sent to our editor, **Andrew Binstock**, at javamag_us@oracle.com.

While it will have no influence on our decision whether to publish your article or letter, cookies and edible treats will be gratefully accepted by our staff at *Java Magazine*, Oracle Corporation, 500 Oracle Parkway, MS OPL 3A, Redwood Shores, CA 94065, USA.

- Subscription application
- Download area for code and other items
- *Java Magazine* in Japanese